



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

TEACHING YOUR WIRELESS CARD NEW TRICKS:  
SMARTPHONE PERFORMANCE AND SECURITY ENHANCEMENTS  
THROUGH WI-FI FIRMWARE MODIFICATIONS

Am Fachbereich Informatik  
der Technische Universität Darmstadt  
zur Erlangung des akademischen Grades eines  
Doktor-Ingenieurs (Dr.-Ing.)  
genehmigte Dissertationsschrift

von

MATTHIAS THOMAS SCHULZ, M. SC.

Geboren am 15. November 1987  
in Wiesbaden-Dotzheim, Deutschland

Erstgutachter: Prof. Dr.-Ing. Matthias Hollick  
Zweitgutachter: Prof. Guevara Noubir (Ph.D.)

Tag der Einreichung: 12. Januar 2018  
Tag der Disputation: 26. Februar 2018

Darmstadt, 2018  
Hochschulkennziffer D17

Verfasser: Schulz, Matthias Thomas

Titel: Teaching Your Wireless Card New Tricks: Smartphone Performance and Security Enhancements Through Wi-Fi Firmware Modifications

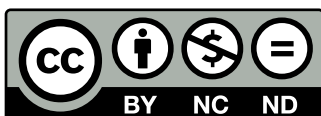
Dissertationsort: Darmstadt, Technische Universität Darmstadt

Jahr der Veröffentlichung der Dissertation auf TUPrints: 2018

URN: urn:nbn:de:tuda-tuprints-72438

URL: <https://tuprints.ulb.tu-darmstadt.de/id/eprint/7243>

Tag der mündlichen Prüfung: 26. Februar 2018



Veröffentlicht unter CC BY-NC-ND 4.0 International

(Namensnennung – Keine kommerzielle Nutzung – Keine Bearbeitung)

<https://creativecommons.org/licenses/by-nc-nd/4.0/deed.de>

Licensed under CC BY-NC-ND 4.0 International

(Attribution – NonCommercial – NoDerivatives)

<https://creativecommons.org/licenses/by-nc-nd/4.0/deed.en>

## ABSTRACT

---

Smartphones come with a variety of sensors and communication interfaces, which make them perfect candidates for mobile communication testbeds. Nevertheless, proprietary firmwares hinder us from accessing the full capabilities of the underlying hardware platform which impedes innovation. Focusing on FullMAC Wi-Fi chips, we present Nexmon, a C-based firmware modification framework. It gives access to raw Wi-Fi frames and advanced capabilities that we found by reverse engineering chips and their firmware. As firmware modifications pose security risks, we discuss how to secure firmware handling without impeding experimentation on Wi-Fi chips. To present and evaluate our findings in the field, we developed the following applications. We start by presenting a ping-offloading application that handles ping requests in the firmware instead of the operating system. It significantly reduces energy consumption and processing delays. Then, we present a software-defined wireless networking application that enhances scalable video streaming by setting flow-based requirements on physical-layer parameters. As security application, we present a reactive Wi-Fi jammer that analyzes incoming frames during reception and transmits arbitrary jamming waveforms by operating Wi-Fi chips as software-defined radios (SDRs). We further introduce an acknowledging jammer to ensure the flow of non-targeted frames and an adaptive power-control jammer to adjust transmission powers based on measured jamming successes. Additionally, we discovered how to extract channel state information (CSI) on a per-frame basis. Using both SDR and CSI-extraction capabilities, we present a physical-layer covert channel. It hides covert symbols in phase changes of selected OFDM subcarriers. Those manipulations can be extracted from CSI measurements at a receiver. To ease the analysis of firmware binaries, we created a debugging application that supports single stepping and runs as firmware patch on the Wi-Fi chip. We published the source code of our framework and our applications to ensure reproducibility of our results and to enable other researchers to extend our work. Our framework and the applications emphasize the need for freely modifiable firmware and detailed hardware documentation to create novel and exciting applications on commercial off-the-shelf devices.

## ZUSAMMENFASSUNG

---

Smartphones sind mit einer Vielzahl an Sensoren und Kommunikationsschnittstellen ausgestattet, wodurch sie optimal für den Einsatz in mobilen Testumgebungen im Kommunikationsbereich geeignet sind. Allerdings hindert uns proprietäre Firmware die vollen Fähigkeiten der zugrunde liegenden Hardwareplattform auszureizen, was Innovationen behindert. Speziell für FullMAC-WLAN-Chips präsentieren wir Nexmon – ein C-basiertes Firmware-Modifikations-Framework. Es ermöglicht den direkten Zugriff auf WLAN-Pakete und spezielle Hardwarefähigkeiten, die wir durch die Analyse von Chips und ihrer Firmware herausfanden. Da Änderungen an der Firmware Sicherheitsfragen aufwerfen, diskutieren wir die Absicherung der Firmwarehandhabung, ohne dabei das Experimentieren mit WLAN-Chips zu beeinträchtigen. Zum Präsentieren und praktischen Evaluieren unserer Forschungsergebnisse, entwickelten wir die folgenden Anwendungen. Die erste behandelt Ping-Anfragen in der Firmware statt sie im Betriebssystem zu beantworten. Sie reduziert den Energieverbrauch und die Verarbeitungszeit signifikant. Danach präsentieren wir eine Anwendung, die Techniken aus dem Bereich softwaredefinierte drahtlose Netzwerke einsetzt, um skalierbares Videostreaming zu verbessern, indem Datenströme mit Anforderungen an Parameter der Bitübertragungsschicht versehen werden. Als Sicherheitsanwendung präsentieren wir einen reaktiven WLAN-Störsender, der eingehende Pakete während des Empfangs analysiert und beliebig geformte Störsignale aussendet. Dazu verwenden wir den WLAN-Chip als Software Defined Radio (SDR) mit freiem Zugriff auf Basisbandsignale. Des weiteren stellen wir einen Störsender vor, der Empfangsbestätigungen an den Sender der gestörten Pakete schickt, damit der Durchsatz von ungestörten Paketübertragungen nicht einbricht. Eine zusätzliche Erweiterung passt adaptiv die Sendeleistung an, je nachdem wie erfolgreich ein Empfänger gestört wurde. Darüber hinaus können wir für jedes empfangene Paket extrahieren, zu welchen Amplituden- und Phasenverschiebungen die drahtlose Übertragung eines Pakets geführt hat. Durch Nutzung dieser Informationen und der Fähigkeit zum Ändern von Basisbandsignalen haben wir einen verdeckten Kanal auf der Bitübertragungsschicht entwickelt. Er versteckt Daten in Phasenänderungen von ausgewählten OFDM-Unterträgern. Diese Manipulationen können am Empfänger extrahiert werden. Zudem entwickelten wir einen Debugger, um die Analyse von Firmwaredateien zu vereinfachen. Er läuft als Software auf dem WLAN-Chip und kann Firmwarecode in Einzelschritten ausführen. Um die Reproduzierbarkeit unserer Ergebnisse sicherzustellen und es anderen zu ermöglichen unsere Arbeit zu erweitern, stellen wir den Quellcode von Framework und Anwendungen zur Verfügung. Basierend auf unseren Ergebnissen sehen wir den Bedarf nach frei konfigurierbarer Firmware und detaillierter Dokumentation der Hardware, um zukünftig einfacher neue Anwendungen für WLAN-Chips handelsüblicher Geräte zu entwickeln.

## ACKNOWLEDGMENTS

---

*I thank my supervisor, Prof. Matthias Hollick, for providing a working environment that encourages creativity, for giving me the freedom to choose interesting research topics, and for advising me during the work that resulted in this dissertation.*

*I thank my co-supervisor, Prof. Guevara Noubir, for accepting my request to review my work.*

*I thank my collaborators and the students I supervised for their hard work on implementing ideas that finally ended up in this document. Only by collaborating we were able to achieve our targeted goals.*

*I thank my family, especially my parents, my wife, my sister, and my grandmother for continuously supporting me during my studies.*

*I thank my colleagues for a positive working atmosphere, their support, their humor, their friendship, and all the social events we enjoyed together.*

*I thank our secretary and our admins for their administrative support and the technical infrastructure.*

*I thank both the German Research Foundation (DFG) for funding my work through the Collaborative Research Center (SFB) 1053 MAKI and the Hessian Ministry of Science and Art for funding my work through the LOEWE Research Cluster NICER.*



# CONTENTS

---

<b>I</b>	<b>INTRODUCTION</b>	<b>1</b>
1	MOTIVATION AND GOALS	3
1.1	Problem statement . . . . .	3
1.2	Our approach . . . . .	4
1.3	Our goals and challenges . . . . .	5
2	CONTRIBUTION	7
2.1	Analysis and security enhancement . . . . .	7
2.2	Framework and toolset . . . . .	7
2.2.1	JTAG-less debugging . . . . .	8
2.2.2	Real-time MAC access . . . . .	8
2.2.3	Operating Wi-Fi chips as software-defined radios . . . . .	8
2.2.4	Channel state information extraction . . . . .	10
2.2.5	Position-independent code generation . . . . .	10
2.3	Applications . . . . .	10
2.3.1	Ping-offloading . . . . .	10
2.3.2	Software-defined wireless networking . . . . .	11
2.3.3	Reactive jamming with arbitrary waveforms . . . . .	11
2.3.4	Covert channels by prefiltering outgoing frames . . . . .	11
3	RELATED WORK	13
3.1	Work related to modifying FullMAC firmwares . . . . .	13
3.2	Work related to modifying real-time firmwares . . . . .	14
3.3	Conclusion . . . . .	14
<b>II</b>	<b>CHIP INTERNALS AND FIRMWARE HANDLING</b>	<b>15</b>
4	BROADCOM'S WI-FI CHIPS	17
4.1	SoftMAC vs. FullMAC chips . . . . .	17
4.2	Transmit path . . . . .	19
4.2.1	Physical layer components . . . . .	19
4.2.2	Arbitrary signal transmissions . . . . .	20
4.2.3	Advanced raw signal transmissions . . . . .	20
4.3	Receive path . . . . .	21
4.3.1	Collecting raw samples . . . . .	22
4.3.2	Demodulating Wi-Fi frames . . . . .	22
4.3.3	Frame processing on the receive path . . . . .	23
4.4	Programmable state machine (PSM) . . . . .	23
4.4.1	Programming the PSM . . . . .	23
4.5	Embedded ARM processor . . . . .	24
4.5.1	Flash patching unit . . . . .	24
4.5.2	Debugging core . . . . .	24
4.6	Conclusion . . . . .	25
5	FIRMWARE ANALYSIS AND SECURITY IMPROVEMENTS	27

5.1	Analyzing the current state of firmware handling . . . . .	28
5.1.1	Limitations of the design decisions . . . . .	28
5.2	Improving security in future chip models . . . . .	29
5.2.1	Limiting access to chip internals and memory . . . . .	30
5.2.2	Avoiding flashpatches and restructuring memory . . . . .	30
5.2.3	Restricting debugging of production code . . . . .	30
5.2.4	Hindering static code analysis . . . . .	31
5.2.5	Downsides of delivering encrypted firmware . . . . .	31
5.2.6	Making the signature verification key exchangeable . . . . .	32
5.2.7	The problem with software vulnerabilities . . . . .	32
5.2.8	Randomizing memory allocation on the heap . . . . .	33
5.2.9	Avoiding code execution in data memory . . . . .	33
5.2.10	Handling vulnerability incidents . . . . .	33
5.3	Conclusion . . . . .	34
5.4	My contribution and acknowledgements . . . . .	34
<b>III</b>	<b>FIRMWARE PATCHING FRAMEWORK</b>	<b>35</b>
<b>6</b>	<b>NEXMON FIRMWARE PATCHING FRAMEWORK</b>	<b>37</b>
6.1	Introducing Nexmon . . . . .	38
6.1.1	How to write patches? . . . . .	40
6.1.2	Where to embed the patch code? . . . . .	41
6.1.3	How to patch read-only memory? . . . . .	41
6.1.4	How to side-load functionality into a running chip . . . . .	42
6.1.5	How to analyze the firmware? . . . . .	43
6.1.6	How do dynamically analyze the firmware? . . . . .	44
6.1.7	How to adapt to new firmware files? . . . . .	45
6.2	Achieving testbed goals . . . . .	45
6.2.1	How to handle receptions? . . . . .	45
6.2.2	How to perform transmissions? . . . . .	46
6.2.3	How are frames stored in the firmware? . . . . .	47
6.2.4	How to handle retransmissions? . . . . .	47
6.2.5	How to set transmit powers? . . . . .	47
6.2.6	What are the internal structures? . . . . .	48
6.2.7	How to set channel specifications? . . . . .	48
6.2.8	How to use timers? . . . . .	49
6.2.9	How to transmit arbitrary waveforms? . . . . .	50
6.2.10	How to modulate information onto arbitrary waveforms? . . . . .	51
6.2.11	How to transmit raw signals from Template RAM? . . . . .	52
6.2.12	How to extract channel state information (CSI)? . . . . .	52
6.2.13	How to talk to the firmware? . . . . .	53
6.2.14	How to modify the real-time firmware? . . . . .	54
6.2.15	How to handle SoftMAC chips . . . . .	55
6.3	Discussion . . . . .	55
6.4	Conclusion . . . . .	56
6.5	My contribution and acknowledgements . . . . .	56
<b>7</b>	<b>PROGRAMMABLE FIRMWARE DEBUGGER</b>	<b>57</b>



7.1	Accessing debugging core registers . . . . .	59
7.2	Implementation . . . . .	59
7.2.1	Initializing the debugger . . . . .	60
7.2.2	Preparing to handle breakpoints and watchpoints . . . . .	60
7.2.3	Handling breakpoints . . . . .	61
7.2.4	Handling watchpoints . . . . .	62
7.3	Example application . . . . .	62
7.4	Discussion . . . . .	63
7.5	Conclusion . . . . .	64
7.6	My contribution . . . . .	65
8	CHANNEL STATE INFORMATION EXTRACTOR . . . . .	67
8.1	Implementation . . . . .	67
8.1.1	The size of channel state information . . . . .	68
8.1.2	Pushing channel state information out of the D11 core . . . . .	68
8.2	Experimental evaluation . . . . .	69
8.2.1	Experimental setup . . . . .	69
8.2.2	Analyzing the CSI dumps . . . . .	70
8.3	Discussion . . . . .	71
8.4	Related work . . . . .	72
8.5	Conclusion . . . . .	73
8.6	My contribution and acknowledgements . . . . .	73
9	SOFTWARE-DEFINED RADIOS ON WI-FI CHIPS . . . . .	75
9.1	Implementation . . . . .	76
9.1.1	Raw sample transmission methodology . . . . .	76
9.2	Experiments . . . . .	77
9.2.1	Experimental setup . . . . .	77
9.2.2	Experimental evaluation . . . . .	78
9.3	Discussion . . . . .	79
9.4	Future work . . . . .	79
9.4.1	Controlling Wi-Fi chips from MATLAB . . . . .	79
9.4.2	Comparing Nexmon SDRs to WARP SDRs . . . . .	80
9.4.3	Implementing continuous transmissions and receptions . . . . .	80
9.4.4	New applications on Wi-Fi chips . . . . .	81
9.5	Related work . . . . .	81
9.6	Conclusion . . . . .	82
9.7	My contribution and acknowledgements . . . . .	83
IV	APPLICATIONS . . . . .	85
10	PING OFFLOADING . . . . .	87
10.1	Implementation . . . . .	87
10.2	Experimental evaluation . . . . .	88
10.2.1	Power consumption . . . . .	89
10.2.2	Number of actually transmitted ping requests . . . . .	90
10.2.3	Round-trip times (RTTs) . . . . .	91
10.3	Discussion . . . . .	91
10.4	Related work . . . . .	92

10.5	Conclusion . . . . .	93
10.6	My contribution . . . . .	93
11	SDWNS WITH FLOW-BASED PHY CONTROL . . . . .	95
11.1	Designing a SDWN system for smartphones . . . . .	97
11.1.1	System overview . . . . .	97
11.1.2	Overview of the system components . . . . .	98
11.1.3	Enhancing SVC-video streaming . . . . .	98
11.1.4	Complete system overview . . . . .	99
11.1.5	Isolating the application from physical-layer settings . . . . .	99
11.1.6	Interfacing SDRs from smartphones . . . . .	102
11.1.7	Offering enhanced features at the receiver . . . . .	102
11.1.8	Robust scalable-video transmission . . . . .	103
11.2	Implementation . . . . .	104
11.2.1	Implementing the scalable-video codec . . . . .	104
11.2.2	Implementing the WARP “VPN” Service . . . . .	105
11.2.3	A Nexmon-based implementation using internal Wi-Fi chips . . . . .	106
11.3	Evaluation . . . . .	108
11.3.1	Experiment definition . . . . .	108
11.3.2	Evaluation of transmit rate variations . . . . .	109
11.3.3	Evaluation of transmit power variations . . . . .	110
11.4	Discussion . . . . .	112
11.5	Future work . . . . .	113
11.6	Related work . . . . .	113
11.7	Conclusion . . . . .	114
11.8	My contribution and acknowledgements . . . . .	115
12	REACTIVE WI-FI JAMMING ON SMARTPHONES . . . . .	117
12.1	Design . . . . .	119
12.1.1	Reactive jammer . . . . .	119
12.1.2	Acknowledging jammer . . . . .	120
12.1.3	Adaptive power-control jammer . . . . .	120
12.1.4	Jamming signal generation . . . . .	122
12.1.5	Signal amplification . . . . .	122
12.1.6	Power consumption . . . . .	123
12.2	Implementation . . . . .	125
12.2.1	Jamming app . . . . .	126
12.2.2	Implementation in the D11 core . . . . .	128
12.2.3	Implementing the reactive jammer . . . . .	128
12.2.4	Implementing the acknowledging jammer . . . . .	129
12.2.5	Implementing the adaptive power-control jammer . . . . .	129
12.3	Experimental evaluation . . . . .	130
12.3.1	Experimental setup . . . . .	130
12.3.2	Evaluating our reactive jammer . . . . .	132
12.3.3	Reactively jamming non-compliant 802.11ac transmissions . . . . .	134
12.3.4	Multi-node jamming analysis . . . . .	135
12.3.5	Flow-selective jamming . . . . .	136
12.3.6	Power consumption analysis . . . . .	138

12.4	Discussion . . . . .	139
12.5	Related work . . . . .	140
12.6	Future work . . . . .	142
12.7	Conclusion . . . . .	143
12.8	My contribution and acknowledgements . . . . .	143
13	WI-FI-BASED COVERT CHANNELS . . . . .	145
13.1	Covert channel design . . . . .	145
13.2	Implementation . . . . .	147
13.2.1	Generating and sending acknowledgements with covert information . . . . .	148
13.2.2	Choosing covert symbols . . . . .	149
13.3	Experimental evaluation . . . . .	150
13.3.1	Covert channel experiment in line-of-sight setup . . . . .	150
13.3.2	Evaluating the influence on normal Wi-Fi receivers . . . . .	151
13.3.3	Reception performance at the covert channel receiver . . . . .	152
13.3.4	Choosing suitable symbols . . . . .	153
13.3.5	Real-time experiments involving the D11 core . . . . .	153
13.3.6	Evaluating the experimental results . . . . .	154
13.4	Discussion . . . . .	155
13.5	Related work . . . . .	155
13.5.1	Data-link-layer approaches . . . . .	155
13.5.2	Physical-layer approaches . . . . .	156
13.6	Future work . . . . .	156
13.7	Conclusion . . . . .	157
13.8	My contribution and acknowledgements . . . . .	157
14	PROJECTS USING NEXMON . . . . .	159
14.1	Nexmon for Qualcomm's 802.11ad Wi-Fi chip . . . . .	159
14.1.1	Porting Nexmon to ARC600 cores . . . . .	159
14.1.2	Simplifying debugging of the QCA9500 firmware . . . . .	160
14.2	Nexmon for Fitbit activity trackers . . . . .	161
14.3	Security analyses based on Nexmon's results . . . . .	161
14.4	Nexmon for Qualcomm's LTE modem firmware . . . . .	161
14.5	Conclusion . . . . .	162
V	DISCUSSION AND CONCLUSIONS . . . . .	163
15	DISCUSSION . . . . .	165
16	CONCLUSIONS . . . . .	169
VI	APPENDIX . . . . .	173
A	SOFTWARE RELEASES . . . . .	175
A.1	Nexmon firmware patching framework . . . . .	175
A.2	Ping-offloading application . . . . .	176
A.3	Nexmon Debugger . . . . .	176
A.4	Nexmon Jammer . . . . .	176
A.5	Nexmon SDR . . . . .	177
A.6	Nexmon CSI Extractor . . . . .	177
A.7	Nexmon Covert Channel . . . . .	178

A.8 No-LTE kernel for Nexus 5 . . . . .	178
BIBLIOGRAPHY	179
CURRICULUM VITÆ	187
AUTHOR'S PUBLICATIONS	191
ERKLÄRUNG ZUR DISSERTATIONSSCHRIFT	199

## LIST OF FIGURES

---

Figure 1	Nexmon framework and application overview . . . . .	9
Figure 2	Architecture of a FullMAC Broadcom Wi-Fi chip . . . . .	18
Figure 3	Internals of Broadcom and Cypress FullMAC Wi-Fi chips . . . . .	21
Figure 4	Workflow of Nexmon’s build system . . . . .	39
Figure 5	Experimental setup for CSI measurements . . . . .	70
Figure 6	Waterfall diagrams of CSI measurements . . . . .	71
Figure 7	Experimental setup for SDR experiments . . . . .	77
Figure 8	Capture of Nexmon generated raw transmissions . . . . .	78
Figure 9	Experimental setup of ping-offloading application . . . . .	88
Figure 10	Power consumption for handling pings . . . . .	89
Figure 11	Achievable number of ping requests per second . . . . .	90
Figure 12	Round trip time for answering pings . . . . .	91
Figure 13	Abstract SDWN system overview . . . . .	98
Figure 14	Overview of SVC-based flow-controlled SDWN transmitter . . . . .	100
Figure 15	Overview of SVC-based flow-controlled SDWN receiver . . . . .	101
Figure 16	H.264/SVC cube model . . . . .	103
Figure 17	Experimental video streaming setup . . . . .	107
Figure 18	Smartphone to WARP connection setup . . . . .	107
Figure 19	Frame reception rates of SVC base layer . . . . .	109
Figure 20	Video qualities for fixed transmit powers . . . . .	110
Figure 21	Frame reception rates when keeping the transmit rates fixed . . . . .	111
Figure 22	Video qualities when keeping the transmit rates fixed . . . . .	112
Figure 23	State machine of the adaptive power-control jammer . . . . .	121
Figure 24	PAPR and average power analysis . . . . .	123
Figure 25	PGA and BBMULT settings by index . . . . .	124
Figure 26	Power consumption for different channel specifications . . . . .	125
Figure 27	Power consumption for continuous test tone transmissions . . . . .	125
Figure 28	User interface of the Nexmon jammer app . . . . .	127
Figure 29	Experimental setup in our office building . . . . .	131
Figure 30	Jamming frames at different rates . . . . .	133
Figure 31	Jamming rogue 802.11ac transmissions . . . . .	135
Figure 32	UDP throughputs with either one or two active nodes . . . . .	136
Figure 33	UDP throughputs with two streams sent by one node . . . . .	137
Figure 34	Operation of the acknowledging jammer . . . . .	137
Figure 35	Total power consumption of the jammer . . . . .	138
Figure 36	Covert channel system design overview. . . . .	147
Figure 37	Experimental setup for covert channel evaluation . . . . .	150
Figure 38	Frame reception at regular Wi-Fi receivers . . . . .	151
Figure 39	Covert symbol detection rates . . . . .	152
Figure 40	Covert message reception evaluation . . . . .	154
Figure 41	QCA9500 802.11ad Wi-Fi chip’s memory layout . . . . .	160

## LIST OF TABLES

---

Table 2	802.11g data rates achievable on WARP . . . . .	108
---------	---	-----

## LISTINGS

---

Listing 1	Output of our debugger example application. . . . .	64
-----------	---	----

## ACRONYMS

---

ADB	Android Debug Bridge
AWGN	Additive White Gaussian Noise
CFO	Carrier Frequency Offset
CSI	Channel State Information
DAC	Digital-to-Analog Converter
DCT	Discrete Cosine Transform
DPI	Deep Packet Inspection
DSSS	Direct Sequence Spread Spectrum
FCS	Frame Check Sequence
FEC	Forward Error Correction
FHSS	Frequency Hopping Spread Spectrum
FPGA	Field-Programmable Gate Array
FSK	Frequency Shift Keying
GOT	Global Offset Table
IDCT	Inverse Discrete Cosine Transform
IDFT	Inverse Discrete Fourier Transform
IFFT	Inverse Fast Fourier Transform
IMD	Implantable Medical Device
IP	Internet Protocol
ISI	Inter-Symbol Interference
IV	Initialization Vector
JTAG	Joint Test Action Group
LAN	Local Area Network
LTE	Long-Term Evolution
LTF	Long-Training Field
MAC	Medium Access Control

MCS	Modulation Coding Scheme
MLME	MAC Sublayer Management Entity
OFDM	Orthogonal Frequency-Division Multiplexing
PCIE	Peripheral Component Interconnect Express
PDR	Packet Delivery Ratio
PLCP	Physical Layer Convergence Procedure
PSK	Phase Shift Keying
PSM	Programmable State Machine
PSR	Packet Send Ratio
QAM	Quadrature Amplitude Modulation
QoE	Quality of Experience
RAM	Random-Access Memory
RF	Radio Frequency
ROM	Read-Only Memory
RSS	Received Signal Strength
SDIO	Secure Digital Input Output
SDN	Software-Defined Networking
SDR	Software-Defined Radio
SDWN	Software-Defined Wireless Networking
SHVC	Scalable High-Efficient Video Coding
SSIM	Structural Similarity
STF	Short-Training Field
SVC	Scalable Video Codec/Coding
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UI	User Interface
VoD	Video on Demand
VPN	Virtual Private Network
WARP	Wireless Open-Access Research Platform



WEP	Wired Equivalent Privacy
WLAN	Wireless Local Area Network
WSN	Wireless Sensor Network



## Part I

### INTRODUCTION

We first motivate this thesis and present goals and challenges in Chapter 1. Followed by Chapter 2, where we give an overview of the contributions of this work. In Chapter 3, we present related work.



## MOTIVATION AND GOALS

---

Since Wi-Fi systems emerged at the end of the 20th century, devices using this technology have conquered the world. Today, smartphones and Internet-of-things devices are omni-present. Their success is likely due to their easy, interoperable and free-off-charge wireless access technology to connect to networks—in particular the Internet. Smartphones have definitely changed our lives: they changed how we communicate, how we navigate, how we consume and produce multi media. Using augmented reality, they will also change how we observe our environment. Coupled with Internet-of-things devices that swamp our homes with intelligent light bulbs and other control systems, smartphones become our central remote control and monitoring device, that everyone carries in their pocket.

*Today, almost anyone carries a Wi-Fi enabled smartphone.*

### 1.1 PROBLEM STATEMENT

For both Internet-of-things devices and smartphones, wireless networks are essential. Depending on the application, they either need to cope with high device densities in residential areas or high throughput requirements, for example, to stream video contents. To reduce interference, standards are required that provide backwards compatibility to support legacy devices but also keep up with current state-of-the-art technology. It is important to increase throughput while simultaneously becoming more efficient regarding shared resource usage such as spectral bandwidth and time.

*While demands on wireless networks grow, standards ensure the compatibility of legacy devices.*

While the commitment to standards is important to ensure compatibility and fairness between wireless devices, it also restricts research to boundaries laid out by standard committees. Researchers who intend to explore the full capabilities of their devices to, for example, optimize access to the wireless medium or to integrate new functionalities, generally have to use custom hardware. For wireless systems, this could be software-defined radios. They offer direct access to baseband signals so that researchers can implement new physical layers or enhance reference implementations of standards. Due to field-programmable gate arrays (FPGAs) and exchangeable radio front-ends, software-defined radios offer the highest flexibility for modifying the physical layer. Nevertheless, those devices are generally larger, more expensive and more energy consuming than integrated chips. This limits their applicability in mobile testbeds. Especially in scenarios that focus on cross-layer evaluations, where new

*To evaluate new physical-layer schemes, researchers often use software-defined radios that are flexible development platforms but limited in mobility.*

physical layers should be tested in real-world environments with real applications and user interaction.

*Why not use whatever the hardware offers?*

For many scenarios, however, researchers could enhance wireless communication protocols on the lower layers by simply exhausting the capabilities of a communication device beyond the restrictions imposed by communication standards or the device's firmware. For example, smartphones can be operated as client in a network (managed mode), span an access point (AP mode) or directly communicate with other stations (direct or ad hoc mode). However, they lack the ability to span mesh networks between devices, which would offer decentralized communication capabilities over a large area which could be helpful in emergency situations. To evaluate those mesh networks in practice, researchers are generally limited to devices that allow modifications of data-link layer implementations. Those devices either implement the MAC sublayer management entity (MLME) in open-source drivers (SoftMAC approach) or offer direct access to raw Wi-Fi frames so that they can be received and injected by the operating system or a user-space application. Unfortunately, most (as of today maybe all) smartphone Wi-Fi chips follow the FullMAC approach. Those chips encapsulate all Wi-Fi processing functions in a proprietary firmware file and act like an Ethernet-to-Wi-Fi bridge to the host. Hence, it is normally not possible to change the Wi-Fi frame handling and implement new communication schemes, without access to the firmware source code to rebuild the Wi-Fi firmware file.

*FullMAC Wi-Fi chips in smartphones are hard to adopt to new communication approaches due to the lack of open-source firmwares.*

## 1.2 OUR APPROACH

To circumvent this limitation, we reverse engineered essential parts of the firmware running on Broadcom Wi-Fi chips. Those are used in a high number of smartphones. As a result of this process, we found and named functions according to their tasks. This is the basis for analyzing how frames are handled by the firmware, how they enter and leave the chip and what processing steps are performed in between. The gained knowledge allowed us to modify firmware operations by writing patches in Assembler and to apply them to the firmware file. The lack of signature checks allowed us to execute modified firmware files on smartphone Wi-Fi chips. Writing patches in Assembler soon became tedious and error prone especially to implement more complex projects. Additionally, the portability of written code to other firmware and chip versions was limited. Hence, we started working on a way to write patches in C, compile them and patch the resulting binary blobs into the firmware file. The outcome was our Nexmon firmware patching framework. The name is composed of the two syllables "Nex" and "mon", where the first refers to Nexus 5 smartphones on which we started our research and the second refers to the ability to enable monitor mode on this platform. The framework is

*Firmware reverse engineering provides the foundation for generating firmware patches.*

*The name Nexmon refers to monitor mode for Nexus 5 smartphones.*

the foundation for any of the advanced firmware patches we present in this thesis. They extend beyond simple frame processing modifications in the firmware to completely new applications that use undocumented physical-layer and MAC-layer features of Broadcom Wi-Fi chips that we first had to discover.

*Our firmware patches even demonstrate the use of undocumented chip capabilities.*

### 1.3 OUR GOALS AND CHALLENGES

The main goals of our work are:

- To design and develop innovative physical-layer supported applications for smartphones that demonstrate Wi-Fi chip capabilities that exceed regular Wi-Fi operation.
- To provide means for analyzing and manipulating proprietary Wi-Fi firmwares.
- To discuss potential security risks of unrestricted firmware manipulations and how to cope with them.

To achieve these goals, we had to tackle various challenges. A major handicap for our research was the lack of freely available source code for rebuilding the firmware running on Broadcom's Wi-Fi chips. Without it, we first had to elaborately reverse engineer the firmware binaries to understand how the firmware operates. Then, we had to find means to patch the firmware in a way that even supports complex firmware patches. To this end, we decided for writing patches in C and automating the patching process which required a thorough understanding of the linking process and resulted in our patching framework. Without it, we might not have reached the level of building advanced applications into the the Wi-Fi firmware. Another impediment for discovering capabilities that exceed regular Wi-Fi operation was the lack of hardware documentation. All openly available datasheets only contain superficial information about the Wi-Fi chip and lack a proper description of the available components, their register maps and functions. In the following chapter we give a short overview of the components built around the Nexmon framework.

*The lack of source code forced us into time-consuming reverse engineering of the firmware binary.*

*Proper and available chip documentation would have boosted our innovative outcome.*





## CONTRIBUTION

---

As groundwork for developing new applications running on Wi-Fi chips, we first had to understand the chip internals and discuss how to handle the risks of running modified firmware binaries (see Section 2.1). The fundamental contribution of this thesis is the Nexmon firmware patching framework that comes with a toolset that is reusable by various applications. We present an overview of this work in Figure 1. The toolset consists of exemplary firmware patches that either demonstrate how to use undocumented capabilities we discovered during our research of the Wi-Fi chip, or tools we developed to enhance firmware analysis and operation. We present our toolset together with our framework in Section 2.2. Based on our discoveries, we developed applications to demonstrate and evaluate the capabilities of modern Wi-Fi chips in the field. In Section 2.3, we give an overview of these applications.

*After presenting chip internal and discussing firmware handling, we present our firmware patching framework and exemplary applications.*

### 2.1 ANALYSIS AND SECURITY ENHANCEMENT

Before modifying firmware binaries, a basic understanding of the underlying hardware is required. We present our findings on the chip internals in Chapter 4. The presented information exceeds the level of detail found in official datasheets such as [84]. We especially focus on the frame processing paths in the transmit and receive directions and present hardware components we use for our toolset and the exemplary applications that we present below. As firmware running on Wi-Fi chips has full control over the hardware and the exchanged data streams, users need to trust their device’s manufacturer to act in their best interest and avoid including any malicious code. In Chapter 5, we discuss security implications and propose improvements to the way Broadcom does firmware handling.

*Our chip internals provide the background for firmware patching.*

### 2.2 FRAMEWORK AND TOOLSET

Research into proprietary firmware requires tools to modify and extend firmware binaries. Those tools should let developers focus on the application in mind and abstract from patching details. To this end, we developed the Nexmon firmware patching framework we present in Chapter 6. It consists of a toolchain that compiles patch code written in C into binaries that are automatically integrated into the original firmware file according to annotations in the C code. While this approach works for various hardware platforms (as de-

*Using our framework, developers build firmware patches in C and easily integrate them into the firmware binary.*

*We provide monitor mode and frame injection patches for most of our targeted chips.*

scribed in Chapter 14), we focus on Broadcom’s FullMAC Wi-Fi chips. For most of the supported chips of our publicly accessible repository (see Appendix A), we at least provide a patch to activate monitor mode and frame injection. This is a good base for more advanced firmware patches. In the sections below, we present our toolset that eases the analysis of firmware binaries and the development of more complex applications.

### 2.2.1 JTAG-less debugging

*Dynamic debugging helps to understand disassembled code and is not only available through JTAG.*

Analysing code is often simplified by dynamic observations of the processor state and traces of command executions. To this end, debuggers are used. Most embedded systems offer a JTAG port to connect an external debugger. However, off-the-shelf devices often lack a way to connect to this port. This is especially the case for space constrained devices such as smartphones. To cope without JTAG access, we developed a firmware patch that implements a debugger directly in software. It uses the same ARM debugging core also accessed over JTAG, but triggers debug exceptions that can be handled in firmware instead of completely halting the chip. We present our debugger in Chapter 7. Its support for single-step debugging helped us to generate traces during runtime and thereby understand how the firmware works internally.

### 2.2.2 Real-time MAC access

*Modifications of the Wi-Fi chips real-time behaviour are essential for various advanced applications.*

*Firmware compression makes space for new patch code.*

FullMAC Wi-Fi chips consist of two processors. The first one is an ARM processor that performs tasks that are not time critical. On SoftMAC chips, those tasks are implemented in the driver. Nexmon’s C-based patches only modify the firmware running on this ARM processor. The second processor implements a programmable state machine (PSM). It runs in the D11 core that is responsible to quickly process MAC-layer events. The core decides which frames should be dropped or which should be answered by acknowledgements. Means to modify the code in Assembler already existed for SoftMAC chips and we integrated them into the Nexmon framework for FullMAC chips, as presented in Chapter 6. As the D11 firmware is stored in the ARM firmware binary, extensions require additional free space that we gain by introducing firmware compression. Modifications to the real-time code are essential for applications such as reactive jamming that we present below.

### 2.2.3 Operating Wi-Fi chips as software-defined radios

Most modern communication devices use digital baseband signals for signal modulation and demodulation. Nevertheless, only software-

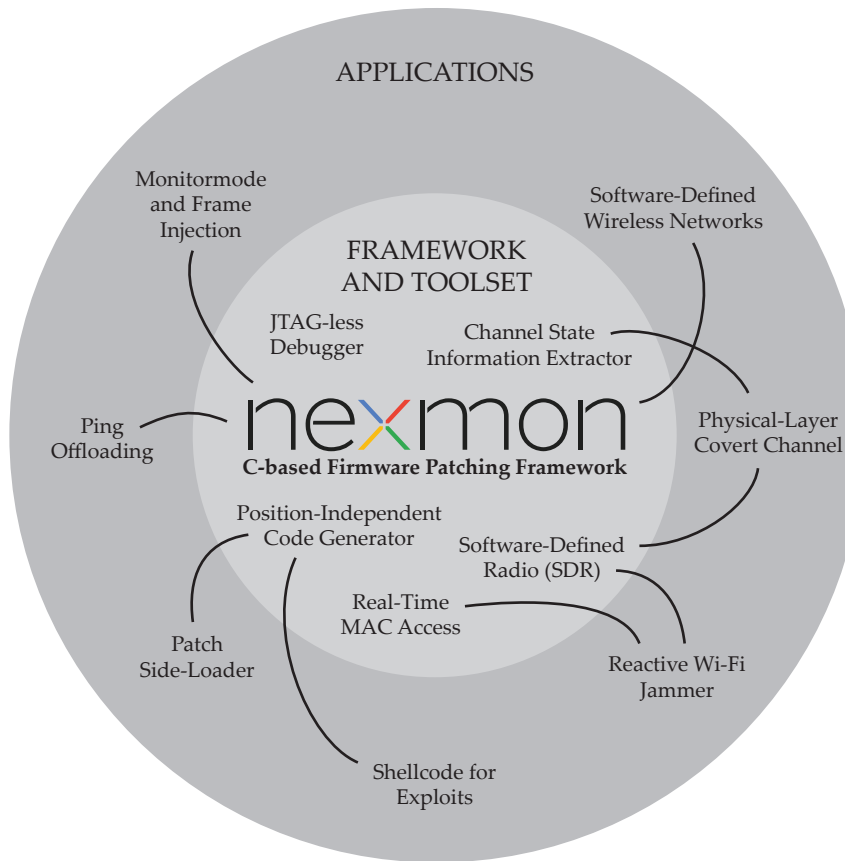


Figure 1: Toolset and applications based on the Nexmon framework that are addressed in this work.

defined radios (SDRs) are designed to give users access to the raw baseband samples. On the one side, this maximizes the flexibility for modulating signals. On the other side, it is less energy efficient than specialized modulation hardware. Hence, Wi-Fi chips generally contain dedicated hardware modulators. During our research, we found two possibilities to get access to raw baseband signals which lets us operate Wi-Fi chips as software-defined radios. The first approach uses an up to 512 samples large buffer for storing tones used for calibration purposes. We use this buffer to transmit arbitrary waveforms in our reactive jamming application presented below. The second approach allows capturing and transmitting samples from a larger memory using 802.11ac capable Wi-Fi chips. On BCM4358 devices, the memory can hold more than 130 000 samples. In Chapter 9, we present a firmware patch that demonstrates the transmission of Wi-Fi frames from raw samples. Additionally, we use SDR-based transmissions to implement our covert channel transmitter presented below.

*While Wi-Fi chips generally contain dedicated modulation hardware, we found ways to operate on raw samples.*

*130 000 samples at 40 MSps sampling rate equal 3.25 ms of signal.*

#### 2.2.4 Channel state information extraction

*Channel state information is required by various applications.*

In the last couple of years, research into different applications was ignited by gaining the ability to extract channel state information (CSI) using Wi-Fi cards. The information describes how the wireless channel influences phases and amplitudes of symbols transmitted on each subcarrier of an OFDM system. So far, tools to extract CSI existed only for Wi-Fi cards installed in laptops, desktop computers or access points. In Chapter 8, we present the first CSI extractor for smartphone Wi-Fi chips and describe how to extract CSI from physical-layer tables on a per-frame basis. We use the CSI extraction capabilities to implement our covert channel receiver presented below.

#### 2.2.5 Position-independent code generation

*We can side-load code into a running firmware by using a custom interface or by exploiting security vulnerabilities.*

For some application scenarios, we need to load code into a Wi-Fi chip after it loaded the firmware file. One reason could be SDWN applications that need to change the firmware's behaviour on demand, for example, by adding frame processing steps. Another reason is the generation of shellcode that could be used in exploits. After finding a vulnerability that allows remote code execution, we need to pack our desired functionalities into a binary that can be loaded to and executed from arbitrary memory locations. To this end, we extended our Nexmon framework to compile single C-files into self-contained binary blobs that use branch instructions relative to the program counter for calling functions within the C-files. For calling external functions at fixed locations, we branch to addresses stored in an offset table. We provide more details on the side-loading code in Chapter 6.

### 2.3 APPLICATIONS

*Our applications demonstrate what is doable in the field.*

Based on our firmware patching framework, we developed and evaluated more complex applications in the field. We use them to demonstrate what could be achieved by modifying Wi-Fi firmwares and gaining access to undocumented capabilities. By providing the source code of our applications, we ensure reproducibility and let other researchers build on our work.

#### 2.3.1 Ping-offloading

*Mesh applications should profit from firmware implementations.*

A simple firmware patch is our ping-offloading application that we present in Chapter 10. We use it to evaluate energy consumption and delay for receiving, processing, and transmitting frames in the Wi-Fi chip compared to handing frames to the smartphone's operating system for processing. A similar functionality is required in mesh sys-

tems on nodes that have to forward frames. Our results show, that the firmware implementation significantly reduces energy consumption and has deterministically low processing delays.

### 2.3.2 *Software-defined wireless networking*

Extending the ping-offloading idea, we propose to realize software-defined wireless networks (SDWNs) using smartphones. Therefore, we consider a scalable-video streaming application that profits from controlling physical-layer parameters. We describe this application in Chapter 11. Our custom video codec splits videos into three quality layers. The first layer has to be received to decode the video, the other two layers increase the image quality. Hence, it makes sense to enforce robust transmissions of the base layer and high-throughput transmissions for the upper layers. In our software-defined networking approach, we consider the three video streams as flows with requirements that we can map to physical-layer parameters. To this end, we implement flow filters either in the user space of a smartphone that is connected to an external SDR or in the Wi-Fi chip.

*We evaluate physical-layer supported scalable-video streaming with flow filters.*

### 2.3.3 *Reactive jamming with arbitrary waveforms*

The purpose of the reactive jammer, presented in Chapter 12, is hindering a receiver from successfully decoding selected frames. To this end, we first need to start receiving frames ourselves and analyze them during reception (real-time MAC access) to check whether a jamming condition matches. In the case it does, we quickly need to switch from receiving to transmitting a jamming signal. While related work uses Wi-Fi frames as jamming signal, we create arbitrary waveforms and trigger their transmission on demand. To achieve this, we use one of our discoveries—the software-defined radio capabilities. Additionally, we extended our jammer to transmit acknowledgements after jamming a frame to trick its transmitter into believing that its transmission was correctly received. This avoids throttling the transmissions we do not jam. In a second extension, we regularly check whether jamming was successful, by listening for acknowledgement transmissions. In case none are received, we reduce the jamming power to minimize energy consumption of our mobile jammer.

*Our reactive jammer avoids correct frame receptions of selected frames.*

*We offer two novel extensions: the acknowledging jammer and the adaptive power-control jammer.*

### 2.3.4 *Covert channels by prefiltering outgoing frames*

Our second security application is a physical-layer covert channel for Wi-Fi systems. It secretly embeds covert information into outgoing Wi-Fi frames and, thereby, offers a communication channel between two stations that is not visible on first sight. Only if an attacker knows

*Covert channels hide the existence of a communication between two nodes.*

*We need SDR  
transmission and  
CSI extraction  
capabilities.*

how the channel is implemented, he can detect and possibly decode it if no encryption is used to protect the confidentiality of the payload. Researching covert channels helps both individuals who need a covert link to exchange sensible information and victims of espionage that want to uncover such links. Our covert channel, presented in Chapter 13, encodes symbols into phase changes on selected subcarriers of outgoing OFDM-based Wi-Fi frames. By extracting channel state information at a receiver, one can decode the covert symbols. In the following chapter, we continue with the presentation of work related to firmware patching in general.

## RELATED WORK

---

This thesis has a focus on modifying Wi-Fi firmwares of FullMAC chips that we mainly find in smartphones. Additionally, we present various tools and applications that emerged from the ability to modify those firmwares. To keep related work close to the topics it relates to, each tool and application chapter has its own related work section. In Chapter 8, we present work related to the extraction of channel state information (CSI) with commercial Wi-Fi cards and its applications. In Chapter 9, we present work related to software-defined radios (SDRs) and the reuse of regular devices as SDRs, for example, DVB-T dongles. In Chapter 10, we present work related to offloading tasks such as TCP checksum calculations into network interface cards. In Chapter 11, we present work related to software-defined networking and scalable-video coding. In Chapter 12, we present work related to jammers. Especially focusing on jamming strategies, applications and implementations on off-the-shelf devices. In Chapter 13, we present work related to data-link-layer and physical-layer covert channels. In Chapter 14, we present projects built on the Nemxon firmware patching framework, which are, thereby, all related. In this chapter, we focus only on work that relates to Wi-Fi firmware modifications in general. We can group it into two categories: (1) work that applies firmware modifications to FullMAC Wi-Fi chips, and (2) work that focuses on modifying the real-time code running on all of Broadcom's Wi-Fi chips.

*This thesis has separate related work sections for each of the discussed topics.*

*In this global related work chapter, we focus on changes to FullMAC firmwares and modifications to real-time firmwares.*

### 3.1 WORK RELATED TO MODIFYING FULLMAC FIRMWARES

Back in 2012, A. Blanco and M. Eissler presented the first firmware patches for FullMAC Wi-Fi chips at the Ekoparty [11] and Hack.lu security conferences. They were the first to analyze and patch the ARM firmware of smartphone Wi-Fi chips. To communicate with the firmware on iOS devices, they used ioctl messages. Their tools to communicate with the firmware, extract the firmware binary and patch it are available in the monmob repository [10]. At a similar time, O. Ildis, Y. Ofir and R. Feinstein developed the bcmmon [43] firmware patch focusing on Android devices. They presented their work at Recon 2013 [44]. There, they introduced the idea of hooking system calls to imitate a monitor interface to applications. This allows using bcmmon's monitor mode capabilities with unmodified applications running on a system with unmodified Wi-Fi drivers. Nevertheless, both approaches have in common that patches are build based on

*Monmob and bcmmon were the first projects to bring monitor mode and frame injection to smartphone Wi-Fi chips.*

Assembly code, which is tedious and error prone to write. Except of an Assembly listing of the monitor mode hook in Blanco’s paper, both patches are closed source and the authors only published tools to apply binary patches or already patched firmware binaries. In contrast, we publish all of our firmware patches as portable C code and also make our patching framework freely accessible so that other researchers can easily start working on their own firmware extensions. In 2015, we introduced a preliminary version of our Nexmon framework in [79]. It already contained the ability to write firmware patches in C, but was limited to the BCM4339 Wi-Fi chip of Nexus 5 smartphones.

*In 2015 we started working on Nexmon and introduced firmware patches based on C code.*

### 3.2 WORK RELATED TO MODIFYING REAL-TIME FIRMWARES

Orthogonal to the development of FullMAC firmware patches, F. Gringoli and L. Nava worked on reverse engineering the firmware running on the real-time processors of Broadcom Wi-Fi chips. In 2009, they published a first version of the OpenFWWF project [33] for Broadcom’s SoftMAC Wi-Fi cards. It consists of an annotated Assembler source code that implements basic Wi-Fi operations. The knowledge about Broadcom’s Wi-Fi platforms collected for the development of the b43 Wi-Fi driver in the bcm-v4 wiki<sup>1</sup> helped Gringoli et al. to understand the operation of the real-time firmware. Based on the OpenFWWF project, various applications emerged. In 2010, Han et al. published their work on partial packet recovering in [35]. In 2012, Tinnirello et al. developed a wireless MAC processor implemented in the real-time firmware in [87]. In 2014 and 2016, Berger et al. published their work on implementing a reactive Wi-Fi jammer using off-the-shelf routers in [8, 9]. During our work on Nexmon, we realized that FullMAC chips contain the same real-time MAC processor as SoftMAC chips. With the help of F. Gringoli, we discovered the location of this firmware within the ARM firmware binary and integrated means into the Nexmon framework to automate the extraction and repacking of this firmware. As a result, all the advanced firmware modifications for SoftMAC chips are now also possible on FullMAC chips.

*F. Gringoli shed some light on how Broadcom’s real-time processor works.*

*We cooperated with F. Gringoli to bring modifications of the real-time firmware to smartphones.*

### 3.3 CONCLUSION

Nexmon extends the idea of bringing monitor mode and frame injection to smartphone Wi-Fi chips to an open source firmware patching framework that is extendable and that allows researchers to build on each others results. By integrating the ability to modify real-time code, we make it possible to bring advanced SoftMAC-based firmware modifications to smartphones.

*Nexmon integrates capabilities of existing projects and makes them accessible and extendable.*

<sup>1</sup> bcm-v4 Specifications: <https://bcm-v4.sipsolutions.net/>



## Part II

### CHIP INTERNALS AND FIRMWARE HANDLING

We first present internals of Broadcom Wi-Fi chips in Chapter 4. Then, we analyze the security of Broadcom's approach to handle firmware and we propose how to enhance firmware handling in Chapter 5.



Before modifying the firmware running on a chip, we should get an understanding of the underlying hardware platform. First, to understand how frames traverse the chip from the host's operating system to the Wi-Fi antenna as well as the other way round. Second, to learn about the hardware components we can use to even exceed the capabilities one would expect from a Wi-Fi chip. In this chapter, we mainly concentrate on BCM4339 Wi-Fi chips installed in Nexus 5 smartphones on which we began our research. We illustrate the chip architecture in Figure 2. On other Broadcom<sup>1</sup> Wi-Fi chips, we find similar components that mainly differ in the processor models and version numbers, as well as the number of signal processing chains in case we have a multi-antenna device. Any Wi-Fi chip has at least physical-layer components to process baseband signals and an analog front end to convert those to radio frequency. For real-time MAC processing, there is the D11 core and to exchange frames with the host, there are interfaces such as SDIO or PCIE. FullMAC chips additionally employ an ARM microcontroller that runs a firmware that implements an Ethernet-to-Wi-Fi bridge. During the research of the hardware, we had to face the following challenges: (1) datasheets on the Wi-Fi chips (such as [84]) only contain superficial information about the available hardware components, and (2) the existence of some capabilities is not even mentioned in the datasheets. Hence, we had to reverse engineer the chips and their firmware to gain the knowledge presented in this chapter.

In the following section, we present the difference between SoftMAC and FullMAC chips, followed by the operation of the transmitter in Section 4.2, the operation of the receiver in Section 4.3, the programmable state machine in Section 4.4, and the embedded ARM processor in Section 4.5. Then, we conclude in Section 4.6.

#### 4.1 SOFTMAC VS. FULLMAC CHIPS

There are two classes of Wi-Fi chips: FullMAC chips and SoftMAC chips. While SoftMAC chips handle non-time-critical tasks in the Wi-Fi driver running on the host system, FullMAC chips move these responsibilities to an embedded processor in the Wi-Fi chip. In the case of Broadcom chips, it is either an ARM Cortex-M3 or Cortex-R4 core.

*An understanding of the chip architecture is required to create advanced firmware patches.*

*Every Broadcom Wi-Fi chip contains similar components for frame and signal processing.*

*We present the different hardware components in detail.*

*FullMAC chips contain ARM microcontrollers.*

<sup>1</sup> In this thesis, we generally refer to Broadcom Wi-Fi chips, even though some of them are now managed and sold by Cypress that acquired Broadcom's Internet-of-things business in 2016.

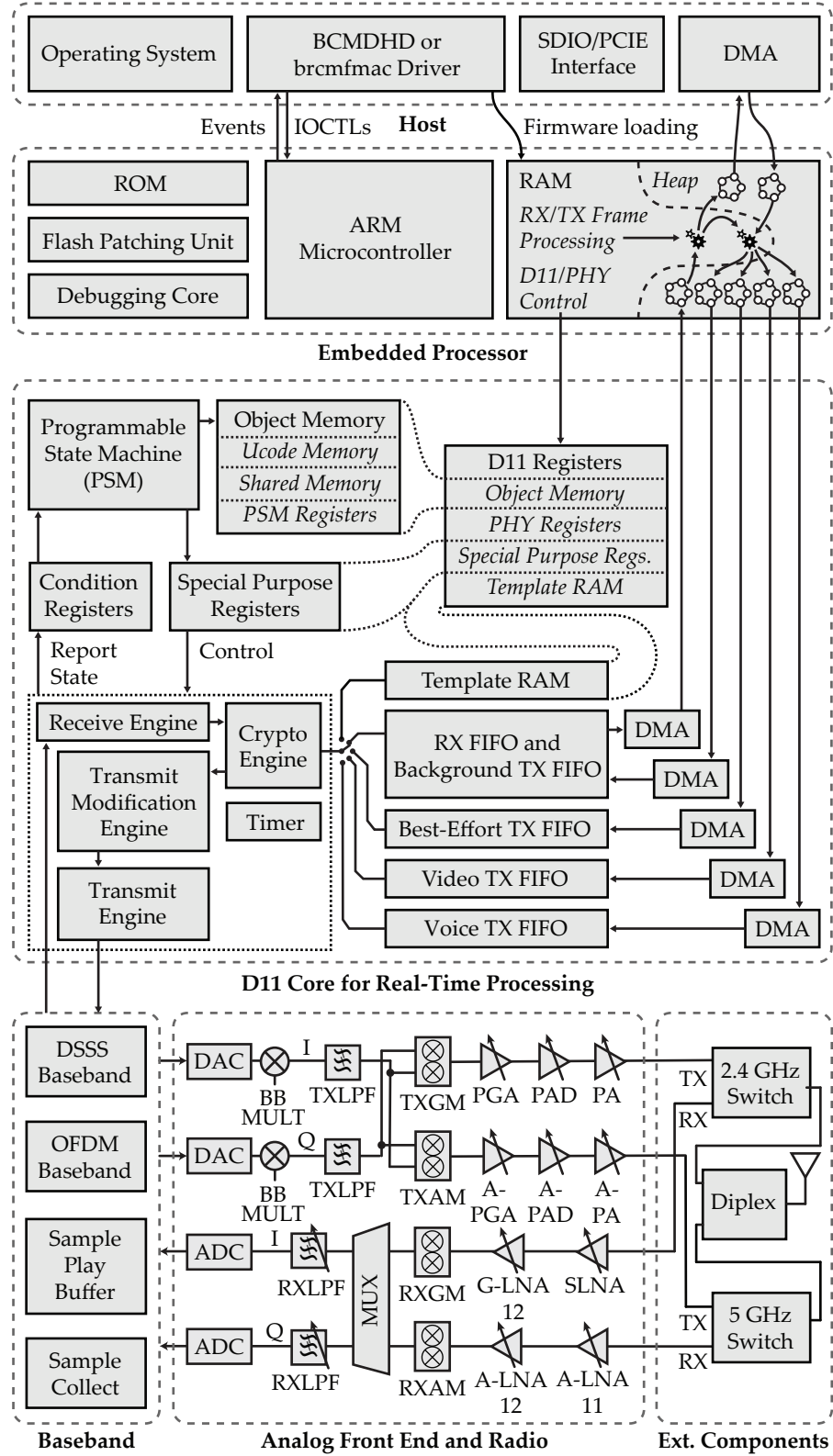


Figure 2: Architecture of a FullMAC Broadcom single-stream Wi-Fi chip, such as the BCM4339 of the Nexus 5. (based on [74, 77])

By offloading tasks such as management and control frame handling into the Wi-Fi chip, energy consumption of the whole device can be reduced. The host processor can stay in a sleep state until the Wi-Fi chip sends application traffic. The frames exchanged with the host are Ethernet frames so that we can consider the firmware running on the Wi-Fi chip as an Ethernet-to-Wi-Fi bridge. Though, FullMAC chips have many benefits, they also restrict direct access to raw Wi-Fi frame processing by design and abstract from MAC and physical layer implementations in the chip. This limits the researchers' abilities to experiment with the full capabilities of Wi-Fi chips. In this work, we focus on working with FullMAC chips and still access their full capabilities by patching the Wi-Fi firmware.

*While SoftMAC chips give full access to Wi-Fi frames, FullMAC chips act as Ethernet-to-Wi-Fi bridges, abstracting from details of wireless systems.*

## 4.2 TRANSMIT PATH

To exchange Ethernet frames with the host, the Wi-Fi chip uses direct memory access (DMA) controllers. After receiving a frame from the host, the Wi-Fi chip is responsible for forwarding the frame's payload over the wireless interface using Wi-Fi headers and correct physical layer settings to reach the destination node. To this end, the ARM firmware fetches the frames from DMA ring buffers and processes them, for example, by removing Ethernet and adding Wi-Fi headers. Then, the firmware places the processed frames into DMA ring buffers intended for communication with the D11 core. Triggering DMA transfers results in moving the buffered frames into FIFO buffers of the D11 core. There, a programmable state machine (PSM) takes over to control specialized frame processing hardware such as the transmit engine that is responsible for passing frames from the FIFOs to the physical layer. Encryption is employed in the crypto engine and frame headers are quickly rewritten in the transmit modification engine. To control these processing steps, the PSM accesses special purpose registers that influence the engines' behaviors.

*All outgoing frames pass the ARM microcontroller to be prepared for a transmission scheduled by the programmable state machine in the D11 core.*

### 4.2.1 Physical layer components

Overall, the D11 core handles all the time-critical Wi-Fi MAC layer tasks. The physical layer consists of mainly three groups of components: the baseband, the analog front end and radio, and the external components. The baseband is responsible for taking the bits from the MAC layer and modulating them into a complex baseband signal. This step is independent of the transmission frequency and only designs the waveforms within the transmission bandwidth, which is either 20, 40, 80 or 160 MHz for 802.11ac systems. Two modulations are available in the baseband, either direct sequence spread spectrum (DSSS) used for legacy 802.11b/g transmissions or orthogonal frequency-division multiplexing (OFDM) used from 802.11a/g

*The baseband performs the digital modulations and demodulations.*

onwards. The output is always a quantized and sampled digital signal that needs conversion into an analog signal by using the digital to analog converters (DACs) of the analog front end.

*Up and down conversion to and from the transmission frequency is performed by direct conversion transceivers.*

The resulting analog signal has a certain maximum amplitude that is limited by the DACs specifications. To amplify the analog signal, the chip offers a baseband multiplier that linearly increases the signal's amplitude according to the multiplication factor `BBMULT`. Then the signal is low-pass filtered (in the `TXLPF`) and injected into quadrature modulators (`TXGM` for the 2.4 GHz band and `TXAM` for the 5 GHz band). Those modulators perform a direct conversion to the transmission frequency defined by the selected Wi-Fi channel numbers. Then the chip uses a series of variable gain amplifiers ((A-)PGA, (A-)PAD, and (A-)PA) to increase the signal power before passing the signal to the external components. Those components pass the 2.4 and 5 GHz signals into a Diplexer to connect both bands to a dual-band antenna. To avoid having separate transmit and receive antennas, switches are used in each signal path to switch between transmission and reception. Arriving at the antenna, frames generated in the operating system finally reach the air and are transmitted to other stations.

#### 4.2.2 Arbitrary signal transmissions

*The Wi-Fi chip can play arbitrary signals from small IQ-sample buffers.*

Besides generating Wi-Fi frames in the baseband, Broadcom's Wi-Fi chips also contain a so called sample-play buffer. It can hold a limited number of either 256 or 512 IQ-samples depending on the chip version. The samples are directly injectable into the analog front end's digital-to-analog converters (DACs). During regular operation, the sample-play buffer is used for calibration purposes. To this end, the ARM firmware uses a CORDIC function generator to place sine and cosine functions at variable frequencies in the sample-play buffer. Then the firmware triggers a playback. The playback loop is either limited in number of repetitions or it may continue indefinitely. Normally, the played signals are supposed to stay in the chip in a loop-back setup. However, by disabling clipping detection, we observed that played signals are sent to the antenna and thereby broadcasted. We use this feature for our reactive jammer implementation in Chapter 12.

#### 4.2.3 Advanced raw signal transmissions

*802.11ac Wi-Fi chips can transmit large portions of IQ samples from Template RAM.*

Besides transmitting samples using the sample-play buffer, Broadcom chips with support for 802.11ac operation also allow transmitting raw IQ samples stored in Template RAM, as illustrated in Figure 3. This memory holds 131 072 IQ samples on a BCM4358, which is sufficient for over 3 ms of raw signal. We can trigger the transmis-

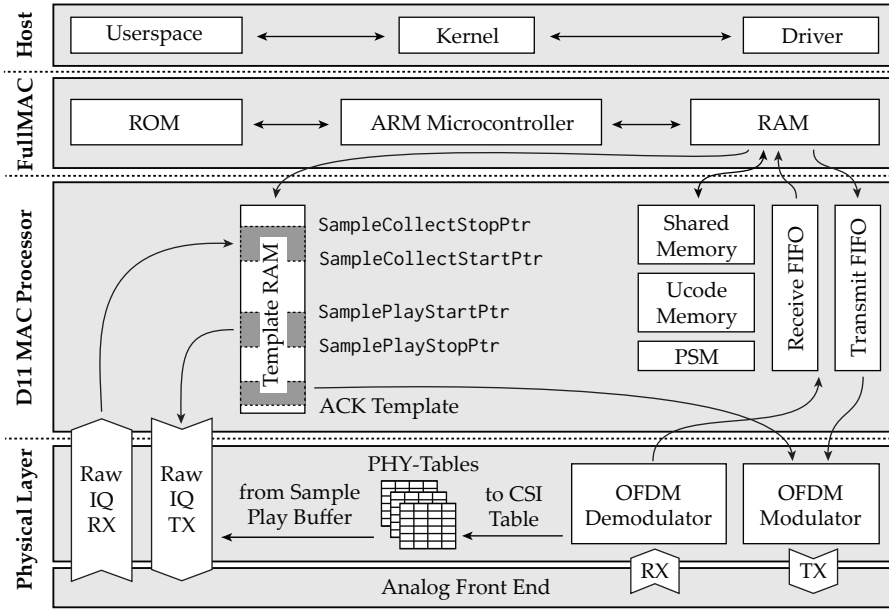


Figure 3: Internals of Broadcom and Cypress FullMAC Wi-Fi chips illustrating regular frame transmit and receive paths as well as raw signal handling capabilities. (based on [75])

sion of raw signals from Template RAM by writing to the D11 core's `psm_phy_hdr_param` register. Using the `SamplePlayStart` and `StopPtr` registers, we can define where raw samples are stored for transmission. Using these registers, we can operate off-the-shelf smartphones in a similar fashion as WARP SDRs [60] running WARPLab, where raw signals are processed in MATLAB on a computer and exchanged through sample buffers on the SDR. We further explain the SDR functionalities in an application example in Chapter 9. In Chapter 13, we further implement a Wi-Fi-based physical-layer covert channel using the SDR transmission features.

*We could generate wireless signals in MATLAB and transmit them using Wi-Fi chips.*

### 4.3 RECEIVE PATH

The receive path starts at the dual band antenna that receives signals and passes them to the diplexer where they are separated by the currently selected reception band (2.4 or 5 GHz). Then, they pass the signal switches and enter the Wi-Fi chip at its receive ports. Here, low noise amplifiers (SLNA, G-LNA 12, A-LNA 11 and A-LNA 12) increase the powers of the very small received signals. Then, the signals are passed to quadrature demodulators (RXGM for the 2.4 GHz band and RXAM for the 5 GHz band) for direct conversion into the baseband, where they are split into inphase (I) and quadrature (Q) components representing complex signals. Those are band limited by low-pass filters (RXLPFs) to comply with the sampling rates of the

*Down conversion in the analog front end consists of amplification, quadrature demodulation and analog-to-digital conversion.*

analog-to-digital converters (ADCs). The latter produce sampled and quantized digital signals that are handled by the baseband.

#### 4.3.1 Collecting raw samples

*Mainly for debugging purposes, raw signal samples can be stored in Template RAM.*

Analog to transmitting raw samples from Template RAM, we can store received raw samples using the sample collect feature of the Wi-Fi chip. It takes a limited number of samples and stores them in the D11 core's Template RAM, from where we can extract them using the ARM core. Similarly to raw signal transmissions, we can trigger the capture by writing to the D11 core's `psm_phy_hdr_param` register. Using the `SampleCollectStart` and `StopPtr` registers, we can define where received raw samples are stored. This feature is most likely meant for debugging purposes and it is questionable whether continuous signal recordings are possible as direct read access to the Template RAM from the ARM core is too slow to cope with the amount of generated samples. Especially as the ADCs sample with twice the channel bandwidth, for example, with 40 MSps when a bandwidth of 20 MHz is selected.

#### 4.3.2 Demodulating Wi-Fi frames

*The effect of the wireless channel on Wi-Fi signals is reflected in the channel state information.*

Instead of collecting raw samples, the physical layer normally demodulates Wi-Fi frames. In case of orthogonal frequency division multiplexing (OFDM)-based Wi-Fi systems, the physical layer first needs to detect the presence of a frame and correlate with the long-training field of the preamble to find the frame's exact starting point. This is required to separate the received OFDM symbols. They are demodulated by applying a fast Fourier transform (FFT) to extract quadrature amplitude modulated (QAM) symbols for each Wi-Fi subcarrier. Due to fading on the wireless channel, amplitudes and phases change between transmitted and received symbols for each subcarrier. To reverse this effect, the physical layer needs to estimate amplitude and phase changes by dividing the received long-training field symbols by their known transmitted equivalents for each subcarrier. The result is called channel state information (CSI) that is inverted and applied to the received data symbols to extract the transmitted symbols. The CSI is stored in so called physical-layer tables, which are memory regions in the physical-layer core accessible by a table identifier and an offset (illustrated in Figure 3). We show how to extract CSI on a per-frame basis in Chapter 8 and use the developed CSI extractor in Chapter 13 as covert channel receiver. The physical layer demodulates the extracted symbols and stores the resulting bytes in the RX FIFOs where they can be processed in real time by the D11 core's firmware (ucode).



### 4.3.3 Frame processing on the receive path

The receive engine controls the receive path and allows to, for example, drop frames, if they are faulty or not required by a node. Additionally, it reports the state of the reception process to condition registers that are, for example, set whenever a frame is detected, the PLCP is completely received or the frame reception finishes. The PSM is designed for quickly reacting to condition changes and controlling the frame handling engines through special purpose registers. From the receive FIFOs, the frames are picked up by direct memory access controllers (DMAs) that perform transfers into the RAM of the embedded ARM processor. There, the frames are stored in ring buffers. The ARM firmware handles received management and control frames internally and rewrites Wi-Fi with Ethernet headers. Frames destined to the host are again stored in ring buffers for DMA transfers to the SDIO controller or directly into the hosts memory when PCIE is used to interface the Wi-Fi chip.

*Under the control of the programmable state machine, received frames are stored in the receive FIFO.*

## 4.4 PROGRAMMABLE STATE MACHINE (PSM)

The PSM is based on a Harvard architecture, where code and data memory are split. The so-called ucode or microcode executed on the PSM is stored in ucode memory. The data memory is also accessible by external components and, hence, called shared memory. Both memories are externally accessible through the object memory interface. It also gives access to the 64 PSM registers. The processor was designed to quickly react to changing conditions and, hence, contains many conditional jump instructions that may check a condition registers and jump to handling code in a single instruction. This design allows to efficiently implement a state machine.

*The PSM uses conditional jump instructions to implement the behavior of a state machine.*

### 4.4.1 Programming the PSM

To change the real-time behavior of the Wi-Fi chip's MAC layer, we may change the code running on the PSM. For this purpose, we use the open source b43 (dis-)assembler. It disassembles the ucode and allows reassembly to exactly the same code or an extended respectively changed version of the code. To understand the meaning of the disassembled code, it is helpful to compare it with the code of the OpenFWWF project [33]. Even though, this project is used for older chip generations, the program structure is still similar in 802.11ac chips. To analyze the ucode, we need access to the binary blob that is loaded into the ucode memory during chip initialization. In SoftMAC chips, the ucode is directly loaded by the driver running in the operating system. The ucode can either be included in the driver binary itself or loaded from a file. In FullMAC chips, the ucode is included

*We change the ucode running on the PSM by using the b43 (dis-)assembler.*

*Changing the ucode is essential to modify the chip's real-time behavior.*

in the firmware of the embedded ARM processor. It is loaded during initialization and then the memory is freed and used as heap of the embedded processor. To access the object memory and thereby write the ucode, both driver and ARM firmware write to the D11 registers that give access to ucode memory as well as other MAC and physical layer registers. Changing the ucode is also an essential part of our Nexmon framework described in Chapter 6.

#### 4.5 EMBEDDED ARM PROCESSOR

*RAM and ROM contain the firmware of the ARM processor.*

In FullMAC chips, the embedded ARM processor performs all the non time-critical MAC layer tasks of the Wi-Fi chip and overall acts as an Ethernet-to-Wi-Fi bridge for the host. In each of Broadcom's chips, the ARM processor is accompanied by a ROM and a RAM. The ROM contains one part of the firmware executed on the ARM processor. It does not only contain basic C library functions such as `memcpy` and `printf` but also functions required to interface with the chips hardware, to control the timer, the power management unit, frame allocations, encryption, and various reusable frame handling functions. As the name states, the ROM contents are read only. The RAM on the other hand is writable, even by the driver of the host's operating system. Whenever a Wi-Fi interface is set up, the driver initializes the RAM with the contents of a firmware file and the Wi-Fi chip starts execution directly from RAM. This allows to simply add new functionalities to the Wi-Fi firmware. Some functions required by the ROM code are only available in RAM. To find their locations in each firmware file, the RAM code has a function pointer table at a fixed location that is used by wrapper functions in ROM to call functions that are part of this table.

*While setting up the Wi-Fi interface, the driver loads a firmware file into the RAM.*

##### 4.5.1 Flash patching unit

*Though, the ROM is unmodifiable, flash patches allow to read arbitrary values from arbitrary memory locations.*

For some firmwares, functions stored in ROM that are also called directly by other functions in ROM need to be overwritten to extend them, make them compatible with extended structures or simply patch security holes. As we cannot directly overwrite ROM contents, the Wi-Fi chips offer a flash patching unit. It intercepts read instructions and either delivers up to eight bytes (most chips) or exactly eight bytes (e.g., on the BCM43596ao) from selected RAM locations. Using flash patches, we can change the program flow in ROM on a limited number of locations. Generally, 256 patches are possible.

##### 4.5.2 Debugging core

All Wi-Fi chips with ARM Cortex-R4 processors are also equipped with a debugging core. It allows us to set breakpoints or watchpoints

in the firmware running on the ARM processor. Configuration of the debugging core is either possible through the JTAG interface or by directly writing to memory mapped registers from the ARM core itself. As the JTAG pins are generally not accessible on Wi-Fi chips in smartphones, the latter way to configure the debugger is preferable. It also allows to handle debugging events directly in the Wi-Fi firmware by triggering a debugging exception. We can handle it directly in the firmware using a separate stack to avoid overwriting the original stack, which allows to return execution back to the original firmware code that triggered a breakpoint or watchpoint. Using the debugging core allows detailed analysis of the firmware during runtime, especially by using single-step debugging for analyzing which paths through the firmware are used. In Chapter 7 we present the implementation of such a debugger in detail.

*Even without a JTAG interface, hardware-supported debugging is possible.*

#### 4.6 CONCLUSION

In this chapter, we met our challenges of finding and presenting information missing in the official datasheets. Those findings are valuable for implementing our firmware patching framework described in Part III and applications based on it presented in Part IV. In the next chapter, we analyze how firmware is handled and how to secure this process.

*Our findings help modifying firmwares.*



## FIRMWARE ANALYSIS AND SECURITY IMPROVEMENTS

---

The security of a device directly relies on the security of the software executed on this device. In the case of embedded devices, this software is called firmware and responsible for interacting with a driver in an operating system and the hardware components of the chip. On Broadcom Wi-Fi chips, the firmware is split into two parts. One part resides in read-only memory (ROM) and is installed during production of the chip. The other part is loaded into random-access memory (RAM) by the driver. This is a common approach to allow easy updates of the firmware to either fix security holes or change modes of operation. Those are generally station or access point modes for which separate firmware binaries exist that the driver loads on demand.

As Broadcom's Wi-Fi chips do not verify the validity of a firmware file, we can apply modifications to the firmware binary and reload it into the chip. This is both a blessing and a curse at the same time. As researchers, we like this open nature as it allows to run modified firmwares on off-the-shelf devices. However, from a security point of view it is very dangerous to load arbitrary code on communication devices, as they could be reconfigured for malicious purposes. For example, to manipulate or sniff network traffic or even execute physical attacks on the wireless channel. Additionally, the operation of modified wireless hardware is generally illegal without a research or an amateur radio license. Hence, in this chapter, we analyze Broadcom's current firmware handling approach and discuss how it can be improved to both improve security for regular end users but also allow low-level access to off-the-shelf devices for researchers.

Besides hindering users from running arbitrary codes on their chips, Broadcom's firmware binaries also lack various security mechanisms that avoid or at least impede remote over-the-air attacks to succeed. Those were demonstrated by G. Beniamini in [7] and by N. Arstein in [3]. Both projects allow attackers to remotely inject and execute arbitrary code on the Wi-Fi chip. In general, we have to assume that every piece of software with a certain complexity always has security issues. Deterministic memory allocations and disabled security features in the compiler, however, allow to easily exploit those holes to take over control of remote systems. In this work, we also analyse the shortcomings of Broadcom's firmware handling with respect to security vulnerabilities and propose solutions to enhance security.

*Every Broadcom Wi-Fi firmware consists of two parts—one in ROM the other one modifyable in RAM.*

*Broadcom firmwares are freely modifiable by both researchers and attackers.*

*Licenses are required to perform wireless experiments.*

*Nexmon helped security researchers to identify severe code injection vulnerabilities in Broadcom's firmwares.*

Those approaches go hand in hand with the verifications of firmware binaries to avoid malicious code from being executed.

In the following section, we present an analysis of the current firmware handling implementation, followed by Section 5.2 on how to secure the firmware loading procedure and how to reduce risks of remote exploitation.

## 5.1 ANALYZING THE CURRENT STATE OF FIRMWARE HANDLING

As stated above, Broadcom allows to reload firmware binaries from the driver. Depending on the chip model, the embedded processor running the firmware is either an ARM Cortex-R4 or an ARM Cortex-M3 microcontroller. Both contain vector tables at the beginning of the ROM firmware to handle exceptions. Those can be interrupts, errors or the chip's reset. In any case, the table entries point to addresses in RAM. This indicates, that Broadcom always redirects the program flow into RAM, so that any code loaded there is always executed without any further checks. The ROM, on the other side, contains code that is valuable for different firmware implementations loaded into RAM. For code running in RAM, calling functions in ROM is easy as the code positions do not change. If ROM code, however, wants to call RAM functions, it needs to know where they were placed by the linker. To solve this problem, there is a global offset table (GOT) at the beginning of the RAM, containing functions pointers. The GOT is always placed at the same location so that its entries can be read and jumped to by wrapper functions in ROM. Sometimes, those pointers also point at function implementations in ROM. If a function implementation needs an update, one simply adds its code into RAM and updates the GOT. For ROM functions not included in the GOT, Broadcom uses a different patching mechanism. As the ROM contents cannot be overwritten, Broadcom employs a flashpatching unit. It inspects every read operation on memory locations and allows to replace the read-back value by up to eight arbitrary bytes. Though the number of such patches is limited, it enables firmware developers to overwrite mainly beginnings of functions in ROM by branch instructions into new function implementations in RAM. This also allows fixing security vulnerabilities in ROM using a firmware update.

*Broadcom's FullMAC Wi-Fi chips are either equipped with ARM Cortex-R4 or Cortex-M3 microcontrollers.*

*The global offset table (GOT) contains function pointers jumped to by wrapper functions in ROM.*

### 5.1.1 Limitations of the design decisions

Even though, the presented design decisions work in practice to offer flexibility for updating firmwares, it also results in various security implications. First of all, the loaded firmware is neither signed nor encrypted. That allows reverse engineering the firmware binary directly with disassemblers and decompilers, modifying it and also running the modified firmware. While this is helpful for researchers aiming

*Unsigned and unencrypted firmwares are prone to analysis and modifications.*

at extending the firmware, it also allows malware developers—who already gained access to the operating system interfacing the Wi-Fi chip—to modify the chip’s behaviour. For example, by converting it into a reactive jammer [74] that can create interference on a large scale, if many devices were affected. Additionally, splitting the firmware in a part running in ROM and another running in RAM may hamper security. As the number of flashpatches is limited, also the number of security patches to ROM code is limited. As demonstrated in [3, 7], security holes exist that even allow remote code execution. The fact that such holes were found in the firmware implies that it is likely that even more security holes exist. As soon as the number of flash patches required for closing those holes exceeds the number of available flash patches, certain holes will never be fixed which leads to a high number of vulnerable devices that are still in use. In the next section, we describe how to cope with the problems presented above.

*To avoid running out of flashpatches or abusing them, the whole firmware should fit into RAM to make flashpatches superfluous.*

## 5.2 IMPROVING SECURITY IN FUTURE CHIP MODELS

To solve the presented security problem, a redesign of the firmware loading mechanism and firmware storing implementation is required. This is not possible to perform in the currently existing chips as their architecture offers unrestricted memory access from the driver and no abilities to perform a firmware validation in a secure environment. To hinder adversaries from loading modified firmware binaries, each firmware loaded into RAM should be digitally signed. The chip needs to first verify the correctness of the signature, before executing the loaded firmware code. To avoid modifications to or bypassing of the signature verification code, it has to be stored in the chip’s ROM, which at least requires newly produced chips with different ROM contents. Additionally, the vector table entry used for handling a chip reset must not point to a location in RAM. Instead, it has to point at a bootloader code in ROM that is also responsible for checking the validity of the loaded RAM firmware. Only after validating the code in RAM, the program flow may be directed into this code. The signature check itself must be based on an asymmetric cryptographic algorithm for which only the public key is stored in memory. Any code executed on the Wi-Fi chip should never be able to overwrite this key. A symmetric cryptographic algorithm is not sufficient in this case, as it has to be assumed that an adversary is able to extract the key stored in the chip. This would allow him to sign his own firmwares which compromises the security of the firmware validation.

*Execution needs to begin from ROM that contains trusted code.*

*The ROM should only contain a bootloader that verifies firmwares loaded into RAM.*

### 5.2.1 *Limiting access to chip internals and memory*

*To avoid firmware modifications during runtime, direct memory access needs restrictions.*

To load the firmware into RAM, the host system needs direct access to this memory. However, after loading and directly before verifying the firmware, the host system must be locked out from any direct access to the chip internals. From this point on, only the verified firmware should be able to read from and write to chip internal memory. Otherwise, the host system could modify the firmware after being verified. To exchange frames and control commands between Wi-Fi firmware and host system, a direct memory access controller should be used that checks memory access policies and, thereby, avoids unauthorized memory access.

*The host should not have direct access to the MAC and PHY layer registers.*

In addition, the host system also needs to be hindered from accessing any chip internal registers directly. Otherwise, an adversary could skip loading any Wi-Fi firmware and directly talk to the MAC and PHY layer components to maliciously access the wireless channel. Alternatively, an adversary could write to the flashpatching registers to overwrite the public key used for signature verification to inject his own key. In the current design, the host system has direct access to the backbone bus that interconnects all the chip internal cores. Hence, this security features requires a redesign of the chip to host interface.

### 5.2.2 *Avoiding flashpatches and restructuring memory*

*Placing all firmware code into RAM renders the flashpatching unit superfluous.*

As stated above, the flashpatching unit could be used to override memory locations if access to its registers is not sufficiently secured. Additionally, the flashpatching unit is only required to allow patching ROM code, but the numbers of patches is also limited. Both problems, the ROM patching limitations and the flashpatching unit itself, can be solved by simply reducing the ROM size to only hold the bootloader for verifying code loaded into RAM and extending the RAM size to hold the complete firmware code. This would allow full flexibility for organizing memory and also reduce the available functions to only those required by the currently loaded firmware. For example, the firmware delivered to customers could be a release build without any debugging outputs, while the firmware used during development could be more open.

### 5.2.3 *Restricting debugging of production code*

Debugging features in general offer reverse engineers a wide attack surface. Besides messages (including function names) written to a chip internal console, Broadcom chips also contain a JTAG interface and access to the ARM internal debugging registers. While the JTAG pins are often not accessible in off-the-shelf systems, an adversary may desolder a chip to connect to these pins to monitor the oper-



ation of a firmware running on the chip. To avoid this attack, the boot loader should make sure that the JTAG interface is disabled before starting to verify a firmware binary, at least on release builds. To debug code by setting breakpoints and watchpoints, an adversary can not only use the JTAG interface but also directly access debugging registers from software. After enabling the internal ARM debugger core on Cortex-R4 microcontrollers, debugging events can be programmed to trigger an exception that is handled by a function listed in the vector table. This allows to handle debugging events directly on the embedded processor and continues execution with low delays after handling an event. For the BCM4339 we implemented an example debugging application that enables automated single-step debugging to analyze the behaviour of the firmware at runtime (see Chapter 7). To hinder reverse engineers from using the internal debugger, Broadcom should make sure that the debugger is disabled and cannot be enabled by software anymore.

*Besides over JTAG, ARM chips also offer direct access to the debugging core from within the firmware to implement and handle debugging events.*

*We developed an example debugging application running in software.*

#### 5.2.4 *Hindering static code analysis*

Disabling debugging options already hinders reverse engineers from analyzing code during runtime, however, it does not avoid static code analysis. To impede the latter, Broadcom should make sure that no one may get direct access to the binary code executed on the Wi-Fi chip. To achieve this, Broadcom could rely on firmware encryption. The firmware could be encrypted using a symmetric cryptographic algorithm. The key needs to be safely stored in the chip as the extraction of this key would allow attackers to decrypt all firmware binaries for this chip. An asymmetric key would not increase security in this case, as its extraction would as well allow firmware decryptions. In the end, this protection only works until a key gets extracted. The past has taught us, however, that even protected keys are extractable by side-channel or chip-reverse-engineering attacks. Additionally, firmware encryption can also be considered as security by obscurity and it simplifies new attack vectors, we present below.

*Static code analysis can be impeded by encrypted firmwares until the decryption key gets extracted.*

#### 5.2.5 *Downsides of delivering encrypted firmware*

Trying to hide the code running on the Wi-Fi chip from analysis might impede adversaries analyzing and manipulating firmwares, but it also requires customers to trust Broadcom as well as their employees handling the firmware source code. Blindly trusting the firmware developers, however, should be avoided. Even if Broadcom has only good intentions, a rogue employee—probably motivated by an intelligence agency—has the ability to embed a malware into the Wi-Fi firmware that may spy on the communication of millions of end users. In case Broadcom wanted to proof that their firmwares were

*Open-source firmwares are required to impede the shipment of malware in stock firmwares.*

clean, it could publish the source code required to compile each publicly available firmware binary and also provide the toolchain to build the byte-wise same firmware binary again. Then independent security analysts could examine the source code, check for inconsistencies and verify that compilation results in the provided binary firmware files. Combined with signature checks in the Wi-Fi chip, Broadcom could hinder users from executing self-build modified firmwares but still offer transparency to the code running on their chips.

#### 5.2.6 Making the signature verification key exchangeable

*Signing any custom firmware with Broadcom's private key can mitigate security on all chips.*

*End users or researchers might want to experiment with custom firmwares on their own devices.*

*Write access to the key should be controlled by physical proximity.*

Certain individuals, such as researchers might like to modify the firmware running on their own device to evaluate how future systems could be improved performance-wise or security-wise by modifying Wi-Fi firmwares. Additionally, it would allow them to apply security patches directly to the firmware source code when a Wi-Fi chip exceeds its planned end-of-life and does not get firmware updates anymore from Broadcom. To be accepted by the bootloader, however, the custom firmware requires a valid digital signature that only Broadcom can produce. As it is not an option to share the private key for signing firmware binaries, we propose to make the public key stored in the Wi-Fi chip exchangeable. If an end user wants to run custom firmwares he should be able to install his own public key and then load self-signed binaries. To avoid malicious apps or remote attacks from changing the key, write access must not be granted through a software interface, neither to the driver nor to the firmware. We propose to store the public key on the chip in a separate rewritable flash memory that offers read-only access from the chip internal bus and write access through a single-wire serial protocol on a free pin. This pin, ground and the flash memory's supply voltage should be accessible at the edge of the Wi-Fi chip's package so that reprogramming is possible even if the chip was soldered to a printed circuit board. This way, a developer needs physical access the device containing the Wi-Fi chip to exchange the key. This requirement authorizes him to change the key. To avoid an adversary who gains access to a device to change the key without the knowledge of the device's owner, the end user should always have the ability to read and verify the installed key using the Wi-Fi driver.

#### 5.2.7 The problem with software vulnerabilities

*Unsigned code may run on a protected Wi-Fi chip after exploiting a software vulnerability.*

Above, we propose a firmware handling mechanism based on signature verifications and physically restricted write-access to the public-key memory to ensure that only valid firmwares from Broadcom or self-signed firmwares can be loaded into the Wi-Fi chip. Unfortunately, this security feature can be circumvented by code injection

vulnerabilities in a Wi-Fi firmware. Hence, it is inevitable to secure firmwares loaded into the Wi-Fi chip against such attacks. The firmware itself runs without an operating system directly in the microcontroller of the Wi-Fi chip. Nevertheless, standard library functions such as `memcpy` and `malloc` exist. Previous work such as [3, 7] has demonstrated that Broadcom's firmware developers, for example, did not always verify the validity of the length parameter passed to `memcpy`. This led to simple buffer overflow vulnerabilities. As these and similar vulnerabilities are unavoidable in large software projects with a large number of developers, security precautions have to be considered to hinder adversaries from exploiting those security holes.

*Buffer overflows should not lead to code execution.*

### 5.2.8 Randomizing memory allocation on the heap

First of all, memory allocation on the heap is predictable and structures created in the initialization phase of the chip always end up at the same position in memory, which simplifies exploits. This could be avoided by placing structures at random addresses in memory but would also lead to a less efficient memory utilization. A larger RAM would compensate for that limitation. A simple form of memory randomization could be employed by a firmware update.

*Deterministic placement of structures on the heap should be avoided.*

### 5.2.9 Avoiding code execution in data memory

Additionally, memory regions containing code should be marked as executable but without write permissions. Regions containing data should have the no-execution bit set. This would hinder adversaries to inject attack code into data memory and trigger its execution. Even though the Wi-Fi chip's microcontrollers support the no-execution bit, Broadcom does not employ this security feature. Rearranging the RAMs memory layout and activating this feature should be possible by employing a firmware update. Even though, the firmware would still be vulnerable to return-oriented programming attacks, the attack surface is heavily reduced.

*Currently, there are no restrictions where executable code may reside in memory.*

### 5.2.10 Handling vulnerability incidents

Though vulnerabilities may be fixed by a firmware update, a downgrade to a legacy firmware may open the vulnerability again that allows the injection of code into an otherwise protected firmware. Attackers could use this feature to circumvent the protection offered by firmware signatures and run their own code on the Wi-Fi chip. To protect users who do not want to execute modified firmwares, such downgrade attacks should be prohibited. One option to achieve this goal is to always store the highest version number found in successfully verified firmware binaries run on the Wi-Fi chip. Loading

*Restricting firmware execution based on version numbers avoids downgrade attacks.*

*Hardware pins should allow users to reset version counters.*

firmwares with lower version numbers should fail. The version number checking should be implemented in the bootloader and only there writable access to the memory location should be possible. Unfortunately, this approach also restricts the execution of legacy Wi-Fi firmwares that might be free of known vulnerabilities. To re-enable the user to load these, the stored version number should be resettable, for example, by using an external pin on the chip as described in Section 5.2.6.

### 5.3 CONCLUSION

*Executing arbitrary firmwares is beneficial for researchers but also poses security problems.*

In this chapter, we discussed Broadcom's firmware handling on Wi-Fi chips and identified the loading of unverified firmware as well as the lack of protection mechanisms against code injection attacks as major security concerns. Simply shutting down the open access to the chip, for example, by signing and encrypting firmwares is not a perfect solution from a security perspective. As long as the firmware stays closed source, end users may never verify that the stock firmware binary is free of any malware. Additionally, open firmwares would ease the creation of security patches even after the estimated end-of-life of a chip. To avoid the unintentional execution of modified firmwares, the binaries should be signed and the signatures should be checked in a bootloader running on the Wi-Fi chip. To allow researchers the execution of modified firmware, we propose a replaceable verification key writable through a hardware interface, that ensures physical proximity before changing the key. Only verifying firmwares before their execution is, however, not sufficient as vulnerabilities during runtime may lead to code injection attacks whose exploitation should at least be impeded. Overall, our solutions enhance the security of firmware loaded onto Wi-Fi chips while still being open for security analysis and research.

*Firmware verification at boot time alone is not sufficient to create a secure environment.*

### 5.4 MY CONTRIBUTION AND ACKNOWLEDGEMENTS

*Evaluating attack vectors on one platform can help to secure another.*

I thank Carsten Bruns for analysing the security architecture of Qualcomm LTE chips and for providing attack vectors in his master thesis [13]. Based on his findings for Qualcomm chips, I came up with propositions for enhancing the security of Broadcom chips extended by the ability to allow researchers to run custom firmware. Additionally, I thank Matthias Hollick for pointing out the needs to make firmware running on end devices more transparent to users and researchers by releasing the source code.

## Part III

### FIRMWARE PATCHING FRAMEWORK

We introduce our Nexmon firmware patching framework that allows to develop patches in C and embed them directly into the Wi-Fi firmware in Chapter 6. In the following chapters, we present our toolset. In Chapter 7, we present a programmable debugger to ease program analysis. In Chapter 8, we present a firmware patch that extracts per-frame channel state information (CSI). In Chapter 9, we present how to implement arbitrary waveform transmissions using Wi-Fi chips. All developed tools are reusable by other researchers.



The wide-spread availability of wireless infrastructure is one of the major factors that lead to the success of smartphones. Their mobility makes them a perfect candidate for mobile testbeds. Also, the Internet of things (IoT) strongly relies on wireless communication for monitoring and control applications. As a small and cheap Wi-Fi-enabled platform, the Raspberry Pi is a good candidate for experimentation in this domain. Both platforms seek for low-energy consumption to enhance battery life. Hence, they use FullMAC Wi-Fi chips to handle Wi-Fi-related tasks in an embedded processor that only wakes up the device's main processor if frames need handling by an application. Unfortunately, FullMAC chips reduce the flexibility to modify Wi-Fi's behavior in testbeds and research applications. To circumvent this limitation, researchers often employ software-defined radios (SDRs) to access lower layers. These modifications would also run on off-the-shelf hardware, but the blackbox nature of FullMAC chips forces researchers to either move to oversized experimental platforms or limit themselves to the capabilities of proprietary Wi-Fi firmware. For patching Wi-Fi firmware, we had to face the following challenges: (1) We needed a solution for implementing complex applications in the firmware, (2) we needed a way to debug code during runtime, (3) we needed to change code in read-only memory, (4) we needed to make space for storing patches in the firmware file and (5) we needed to control the firmware during runtime.

In this chapter, we introduce Nexmon [80], an open-source framework to write firmware patches in C instead of Assembly with a special focus on modifying Broadcom FullMAC Wi-Fi firmwares. Using C as programming language allows rapid prototyping and easy porting of existing algorithms to run on the Wi-Fi chip's embedded processor. By cleverly using linker scripts, we also manage to call functions of the original firmware similar to library functions defined in a header file. We further provide means to free multiple kilobytes of space in the original firmware to place new functionalities.

In Section 6.1, we describe the design and development of the Nexmon firmware patching framework and in Section 6.2, we explain how testbed developers can achieve custom goals. To get a background on the Wi-Fi chip internals, we refer to Chapter 4. In the following chapters we present various application examples based on the Nexmon framework.

*FullMAC Wi-Fi chips are widely available, but their capabilities are limited by firmware restrictions.*

*There is a need for a flexible framework to analyze and modify firmware.*

*Nexmon is our firmware patching framework.*

*Chapter 4 contains background information on Broadcom chip internals required for this chapter.*

## 6.1 INTRODUCING NEXMON

*We use attributes and pragmas to store placement information directly in C code.*

*We use IDA to extract address information from binary firmware blobs.*

*We compress the ucode firmware to gain space for our ARM firmware patches.*

*Our Nexmon GCC plugin extracts attribute and pragma information during compilation.*

*We store function stubs of original firmware functions in the wrapper.c file to define their locations for the linker.*

To create patches for embedded firmwares, we created Nexmon. It follows the philosophy of collecting all the information required for patching a firmware directly in the C files that also contain the patch code. To define where functions and variables (in general symbols) should be placed, we introduced a new `at-attribute` and `targetregion-pragma` that we evaluate during compilation with our plugin for the GNU compiler collection (GCC). This approach allows to reuse Nexmon for patching firmwares of other systems with GCC compiler support. For example, for patching Qualcomm's 802.11ad (millimeter wave) Wi-Fi chip firmwares as described in Section 14.1.

In Figure 4, we present the whole firmware handling workflow. Every firmware analysis starts by extracting both RAM and ROM and analyzing them in IDA to extract address information (see Section 6.1.5) that either ends up in our C patch files to place symbols or in the `definitions.mk` file used to define addresses for patch placement and the location of binary blobs. To make space for our own patch code, we implemented ucode compression based on [49] to roughly half the size of the ucode stored in the ARM firmware. During chip initialization we decompress the ucode directly into the D11 core's ucode memory using an adaptation of Andrew Church's `tiny inflate library`<sup>1</sup> (see Section 6.1.2). Between extraction and compression of the ucode, we can disassemble and extend it. As the binary blob to initialize the template RAM is stored after the ucode, we extract it and let the linker place it directly after the compressed ucode. The space freed by ucode compression is used to store symbols that we do not explicitly place by our `at-attribute`. Instead, we let the linker collect them in a patch-region using our `targetregion-pragma`.

During compilation, our GCC plugin extracts placement information and stores them into a `nexmon.pre` file that Nexmon re-sorts for prioritization resulting in a `nexmon2.pre` file. Then, Nexmon creates linker and makefiles used to produce and embed patch binaries into the original firmware file. To call original firmware functions, we insert their signatures with a dummy function stub and placement information into the `wrapper.c` file. This file is compiled like any other C file, but the resulting binary blobs are not embedded into the patched firmware. Nevertheless, the linker knows where to find firmware functions and is able to call them from our patch code. To avoid redefinitions of all function signatures in a header file, we use the `wrapper.h` file that automatically removes the function stubs and only keeps the signatures. Below, we present how to handle Nexmon in general and in Section 6.2 we explicitly focus on extending Broadcom firmwares.

<sup>1</sup> Original `tinflate.c` file: <http://achurch.org/tinflation.c>



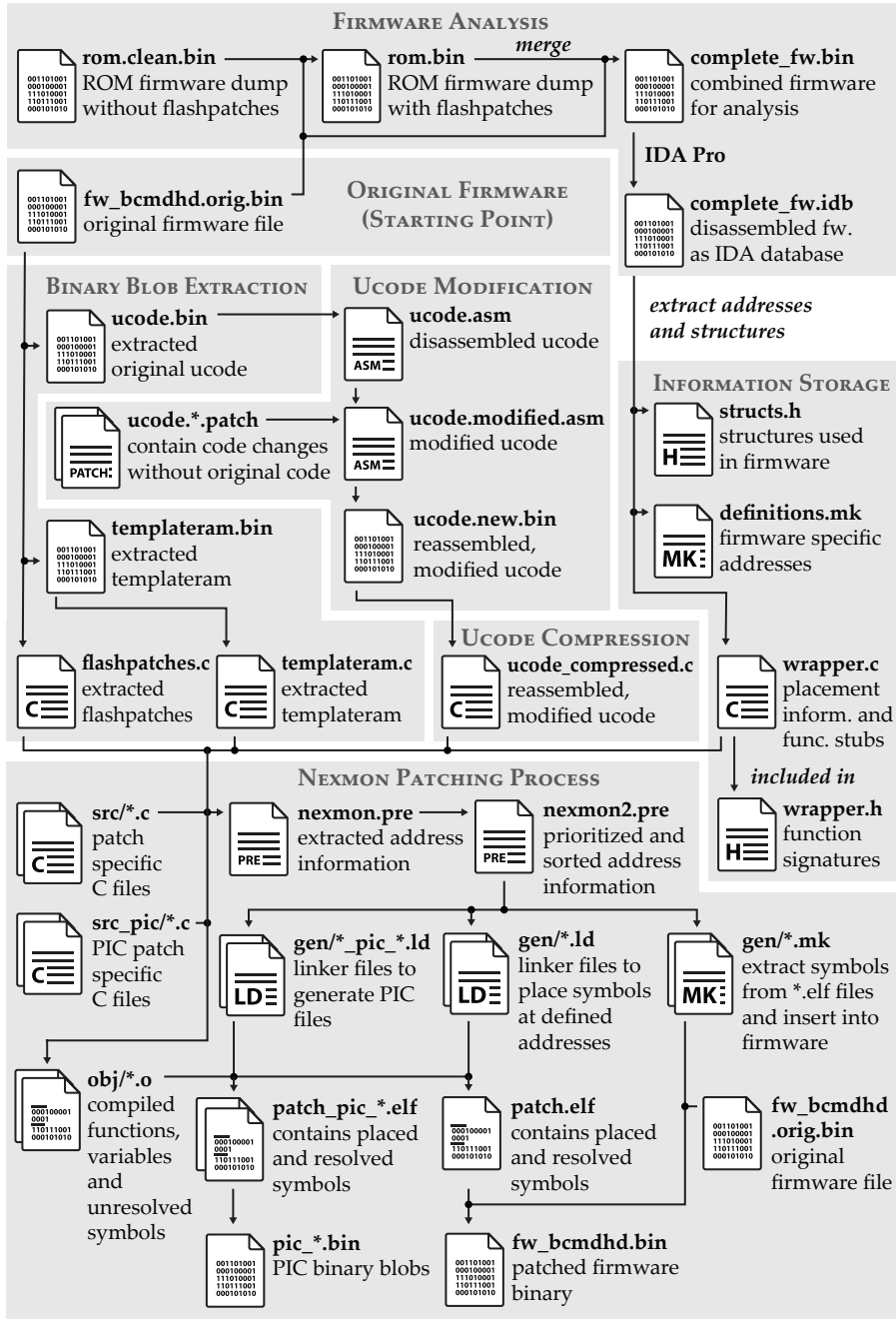


Figure 4: Illustration of the whole Nexmon workflow. We start by analyzing the firmware in IDA to extract address and structure information. Using this information, we extract binary blobs for replacement (templateram), modification (flashpatches) and compression (ucode). We require the latter to attain space for firmware patches. Before compression, we can modify the ucode to change the chip's real-time behavior. To modify the ARM firmware, we write patches in C, link them against firmware functions and merge the result into a new firmware. (based on [77])

### 6.1.1 How to write patches?

To place functions or variables at arbitrary positions, we can prepend their definitions by our `at`-attribute:

```
at(0x100, "", CHIP_VER_BCM4339, FW_VER_ALL)
```

*Our `at`-attribute describes where functions and variables should be placed after linking.*

It takes four parameters. The first defines the target address (e.g., `0x100`), the second is a string that can be set to `"flashpatch"` or `"dummy"`. In `wrapper.c`, `"dummy"` is used to avoid placing function stubs into the firmware. `"flashpatch"` tells Nexmon to create a flash-patch that overwrites up to eight bytes in the ROM at the specified address (see Section 6.1.3). The other two parameters of the `at`-attribute allow to condition the use of this attribute to certain chip and firmware versions (e.g., `CHIP_VER_BCM4339` for the BCM4339 and `FW_VER_ALL` used for symbols in ROM, whose addresses do not change according to the firmware files loaded into RAM). By prepending multiple `at`-attributes with different version parameters, one can write one C file and apply it to multiple platforms and firmware versions.

*We use macros to define commonly used inline Assembly patches.*

Besides simply overwriting a function with a patch function, we supply a set of macros to create patches based on inline Assembly code. They are defined in the `patcher.h` file. Each macro expects a name as first parameter that influences how the generated symbol is called in the linker scripts. Placement is done with the `at`-attribute. Below, we introduce our macros:

*Branch and Branch-Link patches bend the program flow into hook functions.*

`BLPatch(name, func)` and `BPatch(name, func)`: Both create branch instructions resulting in jumps to the target function `func` that can either be a function name or an address. The addresses are calculated relative to the program counter. During runtime, `BLPatch` additionally sets the link register to the address after the created BL instruction which allows to call functions that return.

*HookPatch4 calls a hook function and then returns to the original function.*

`HookPatch4(name, func, inst)`: Calls a hook function `func` before calling the original function by overwriting the first four bytes of the original function with a branch instruction to an intermediate function. The latter pushes the first four registers and the link register to the stack to save them from being overwritten in the hook function `func`. After calling the hook function, this patch pops the saved registers from the stack and executes the instruction `inst` before continuing to execute the original function. The parameter `inst` needs to be the assembler instruction that was overwritten in the original function.

*Generic patches simply replace bytes.*

`GenericPatch1/2/4(name, val)`: Overwrites one, two or four bytes with `val` in the original firmware. We can use the four-byte version to overwrite function pointers in a function table. The target function address should be increased by one to indicate Thumb instruction set.

All symbols, that we do not place explicitly using the `at`-attribute, are collected by the linker and stored in the region defined by the `targetregion-pragma`. For every code file, this should be set to the patch-region that is located at the end of the original ucode blob in the firmware that was freed by ucode compression. Below, we describe how it works.

*The linker collects unplaced symbols in a region defined by our pragma.*

### 6.1.2 Where to embed the patch code?

Symbols that are not explicitly placed are collected in memory regions that also need placement in the firmware file at a location that is not overwritten during runtime. Most firmware files do not have such empty spaces, hence, we needed to find a way to clear space for our patches. Analysing the firmware at runtime, we realized that certain functions and data regions are only needed during the initialization of the Wi-Fi chip. After using the data, the `hndrte_reclaim` function is called to free the now unused space and assign it to the heap. The largest chunk of memory is freed after writing the ucode firmware into the memory of the programmable state machine (PSM) responsible for real-time operations. Analysing this ucode binary reveals that it can be compressed by roughly 50 percent, reducing the size of 44.7 KiB to 22.4 KiB on a BCM4339. This is free space that can be used for our firmware patch code. Hence, we integrated a ucode compression mechanism based on the deflate algorithm into our build toolchain. When the ucode should be loaded into the code memory of the PSM, we decompress it on-the-fly as implemented in the `ucode_compression_code.c` file whose `wlc_ucode_write_compressed` function we call by patching the call to `wlc_ucode_write` in the `wlc_ucode_download` function. To finally reserve the freed space for our patches, we reduced the amount of memory assigned to the heap and placed our patch binaries at the end of the former ucode region. As a side-effect, ucode compression also allows to simply extend the ucode without the need to worry about its size for storing it in the ARM firmware.

*Memory freed after loading the ucode from the ARM firmware into the PSM is assigned to the heap.*

*We compress the ucode to free space for our patches and avoid assignment of this memory to the heap.*

### 6.1.3 How to patch read-only memory?

Besides the firmware that is loaded by the driver into the RAM of the Wi-Fi chip, the chip itself holds a part of the firmware in read-only memory (ROM). Even though, it is not possible to permanently overwrite this part, a flashpatching unit exists in most Broadcom chips. It overlays a number of up to eight byte long memory chunks by data defined in RAM. Reading from those patched locations delivers the overlayed data. Hence, it is possible to redirect the program flow from ROM to RAM by simply overlaying an instruction in ROM with a branch instruction (e.g., by using a `BLPatch` or `BPatch`). Internally,

*Using flashpatches, we can temporarily "overwrite" small locations in ROM.*

*Nexmon extracts existing flashpatches and extends them with those defined in the C patch files.*

flashpatches are defined by creating an entry in the flashpatch configuration array consisting of the target address, the length of the patch and a pointer to the patch data in RAM, which is also stored in an array of eight byte long entries. As the original firmwares do not reserve space to add new flashpatch configurations, we automatically extract all flashpatches and store them in a flashpatch.c file using our fpext utility. During the firmware build, we reassemble the flashpatches and place them into the space freed by ucode compression. After firmware initialization this space is freed and assigned to the heap. To define a flashpatch in C code, one simply uses the keyword “flashpatch” as second parameter of the “at”-attribute: `__attribute__((at(..., "flashpatch", ..., ...)))`

#### 6.1.4 How to side-load functionality into a running chip

*We need reloadable code for extensions during runtime and security analyses.*

Instead of patching the firmware binary once before loading, Nexmon also allows to dynamically reload code during firmware execution. Example applications are the dynamic extension of the firmware according to application requirements (e.g., installing a packet filter after starting tcpdump) or the generation of shellcode that we may inject into a target device by exploiting a remote code execution vulnerability.

*Position independent code (PIC) allows to call local functions relative to the program counter and external functions through a global offset table (GOT).*

To simplify the compilation of such code, we developed the Nexmon PIC extension, where PIC stands for position independent code. With this extension, we can write C files and compile them into separate binary files that are loadable to arbitrary memory addresses from where we can trigger their execution. These files all contain one *main* function that we always place directly at the beginning of our reloadable file. To call the main function, we extended the base firmware to simply branch into the newly loaded code. After the main function, we place additional functions and variables followed by a global offset table (GOT). We need the latter to make the code position independent. As we do not know where the binary blob is loaded during runtime, the code within our blob needs to perform jumps relative to the program counter to reach code within this blob, while existing firmware functions need to be accessed by first loading the absolute target address from the global offset table into a register and then jump to the loaded address.

*We can load firmware extension code through ioctls and make it persistent on the heap of the running firmware.*

To load an executable binary blob, we created an ioctl which is a control message sent from a user-space application or the kernel to the firmware. During this call, the binary blob is loaded onto the heap and stays there until the ioctl processing finishes. Hence, we can directly execute the loaded code by creating a function pointer to its starting address and calling it. If the binary blob should be executable outside the ioctl, for example, in the procedure that processes incoming frames, we first have to copy it into a newly allocated section at

the heap so that it is not overwritten after the `ioctl` processing finishes. Then we can freely pass its starting pointer to other functions to call the new code.

Allowing users to reload code into the Wi-Fi chip during runtime bears potential security risks. If the loaded code is faulty or called assuming incorrect function signatures, the Wi-Fi firmware will likely crash. This is an inconvenience, but the Wi-Fi driver can recover the chip by simply reloading the firmware file. Additionally, we could store a hash of the main function's signature in our binary blob and verify it before branching to the main function. This avoids executing functions with the wrong signatures in case the binary blob is not maliciously modified. By allowing users to inject code into the running Wi-Fi firmware, they gain full control over the Wi-Fi chip. Generally, Android requires users to have root privileges to change the firmware file loaded by the Wi-Fi driver and to send `ioctl`s to the firmware. Hence, everyone able to inject code through `ioctl`s could also directly inject it into the main firmware. Hence, in the default configuration, no additional security risk is introduced. In case, we also want unprivileged users to reload binary blobs, we have to consider that they can gain full access over the Wi-Fi firmware. To limit those users to a selected set of reloadable binary blobs, we can digitally sign those blobs and check their signatures by our firmware patch to ensure that only signed code can be side-loaded into a running Wi-Fi chip by unprivileged users.

*Loading faulty code likely crashes the Wi-Fi chip requiring a restart of the Wi-Fi firmware.*

*Dynamic firmware extensions are a security risk that we can handle by validating function signatures and only executing signed extension binaries.*

#### 6.1.5 How to analyze the firmware?

To analyze the whole firmware binary, the ROM of the Wi-Fi chip needs to be extracted. To extract a clean ROM dump without applied flashpatches, the extraction must take place before the configuration of the latter started during runtime. To achieve this, we created firmware patches that copy the whole ROM content into the RAM directly after starting the chip (`rom_extraction` projects in the Nexmon repository [80]). Then, we wait in an endless loop. To avoid stalling the driver during normal interface setup, we use `dhduutil`'s `download` function to reload the firmware on an already running Wi-Fi chip. Then, we use `dhduutil`'s `membytes` function to dump the RAM content and thereby dump the previously copied ROM contents. To analyze this binary in conjunction with a RAM firmware file, flashpatches should be applied manually to the ROM file using the `fpext` utility.

*A clean ROM dump without applied flashpatches works with any RAM firmware file.*

Equipped with RAM and ROM binaries, we can create a complete binary of the Wi-Fi firmware. To analyze this firmware and find new functions and data structures, we can use IDA Pro with the ARM Decompiler plugin. The latter allows to create C-like code that helps to understand the program flow and allows comparisons to other code sources such as the `brcmsmac` driver that contains functions similar to

*We analyze the combined RAM and ROM firmware file in IDA.*

*The decompiled code can be compared to other code source to get a better understanding of its implementation.*

those in the firmware. In IDA, we first make sure that the code is interpreted as ARM Thumb code in little-endian byte order. Then we start looking for strings that look like function names, find their references and name the enclosing functions accordingly. Then we compare the found function names with functions of the brcmsmac driver or binaries of the wl driver including symbol names to label more functions in the firmware binary. The brcmsmac code also helps to name function arguments and define their types as structures to make the code more readable. Once functions are found and declared in one firmware version, we can use zynamics's bindiff plugin for IDA Pro to find the same functions in other firmwares, even those of other chips.

#### 6.1.6 How do dynamically analyze the firmware?

*Static code analysis is tedious when it comes to understanding what functions do in detail.*

Static code analysis in IDA is a good way to find functions and compare them to available C code. However, to analyze what functions do internally, how they access variables and what values they expect in those to avoid crashing the firmware, we should dynamically analyze the firmware. Using Nexmon, we can patch code for debugging into the firmware. It may write register contents and other debug information to the console using the `printf` function.

*We use the ARM debug core to implement hardware breakpoints and memory watchpoints that generate exceptions that we can handle.*

While this method easily works for hooking functions when they are called, it is more complex when arbitrary instructions within a function need hooks, as some instructions might even be shorter than a branch instruction needed to call the debugging code. To circumvent this problem, we developed a debugger for ARM Cortex-R4 microcontrollers embedded in Broadcom Wi-Fi chips. The debugger uses ARM's debugging code, where we can set up to four breakpoints and four memory watchpoints. As the Wi-Fi chip's JTAG port is generally hard to access in off-the-shelf devices, we decided to run the debugger in monitor mode. That means, whenever a breakpoint is triggered a prefetch abort exceptions is triggered and whenever a watchpoint is triggered a data abort exceptions is triggered. In the standard firmware implementation, the chip simply dumps register contents and parts of the stack to the console and then stops operation when these exceptions occur. In our debugger, we implement our own exception handlers to print debug information and then continue with regular program operation. As we can implement arbitrary logic in those handlers, we end up with a programmable debugger for the Wi-Fi chip's firmware. It even supports single-step debugging that is helpful to understand which path the program takes using the given variable contents. We explain the debugger in more detail in Chapter 7.

*Using single stepping, we can analyze the exact path through the code during execution.*

### 6.1.7 How to adapt to new firmware files?

Each chip has a subdirectory (e.g., BCM4339) under the firmwares directory. Each firmware version has an individual subdirectory (e.g., 6\_37\_34\_43) in such a chip subdirectory. Besides the firmware file (e.g., fw\_bcmdhd.bin), it contains a definitions.mk file with firmware specific addresses, such as the start address and size of the original ucode. To adapt the definitions.mk file, we need to find those addresses in the new firmware mainly by comparing disassembled code pattern of an already analyzed firmware with those of the new firmware. After updating the definitions, we need to find all functions we want to call from our firmware patches. If we already have an IDA file of another firmware version, we can find functions in new firmwares by using IDA's bindiff plugin. After that, we append new "at"-attributes to function stubs in the wrapper.c file containing the addresses in the new firmware. To create a new patching project, it is best to copy one of the nexmon projects from another firmware to the newly added one and adjust all "at"-attributes to place patches at the correct locations in the new firmware file. In the next section, we present how researchers may use the extracted information to achieve goals often required in a testbed but hard to reach with unmodified FullMAC firmwares.

*For each firmware, we need to find ucode, Template RAM and function addresses.*

*Bindiff helps to find functions in new firmware versions again.*

## 6.2 ACHIEVING TESTBED GOALS

Researchers often write firmware patches to accomplish higher goals that are not achievable with unmodified Wi-Fi firmwares. This includes the activation of monitor mode and frame injection to implement custom low-layer communication protocols in the operating system followed by a firmware implementation with reduced latencies and lower power consumption. Besides regular frame processing, Nexmon further offers direct access to the physical layer that, for example, unleashes SDR-like features to transmit arbitrary signals as used in Chapter 12, Chapter 9 and Chapter 13. Below, we present a selected set of goals that can be achieved, mainly focusing on the extension of frame processing capabilities and more control over frame transmission parameters.

*Monitor mode and frame injection are only the most basic features enabled by Nexmon.*

### 6.2.1 How to handle receptions?

In the ARM processor, all frames received by the D11 core are handled in the wlc\_bmac\_recv function that collects them from the DMA ring buffers and passes them to the wlc\_recv function. If monitor mode is active (e.g., by calling nexutil -m1), this function calls the wlc\_monitor function that extracts receive statistics and writes them into the wl\_rxsts structure. Then it passes both the statistics and the

*In monitor mode the firmware calls wl\_monitor to send raw frames to the host.*

frame to the `wl_monitor` function. This is the function we hook to implement monitor mode with radiotap headers. To pass frames to the host, we call the `xmit` function pointer of the interface to the host. On the BCM43430 of the Raspberry Pi, we can even select a separate interface only for monitored frames after creating this interface in the `brcmfmac` driver running on the host.

*The `wl_sendup` function also calls the `xmit` function pointer.*

If the Wi-Fi chip is connected to a network, the `wlc_recv` function also calls a chain of functions used to strip Wi-Fi headers and replace them with Ethernet headers. At the end, `wl_sendup` is called to initiate the transfer of the received frames to the host's operating system. This makes `wl_sendup` the perfect place to implement mechanisms with the benefits of running in the firmware without the need of handling Wi-Fi headers. We use this function in our ping-offloading example in Chapter 10.

### 6.2.2 How to perform transmissions?

*We can change per-frame transmission parameters by editing the `d11txhdr` struct prepended to each frame.*

If connected to a network, we can trigger the transmission of Ethernet frames, for example, after processing a received frame in `wl_sendup`. To this end, we call the `wlc_sendpkt` function. It strips the Ethernet headers, adds Wi-Fi headers and chooses physical-layer parameters required to reach the destination. Responsible for actually setting those parameters is the `wlc_d11hdrs_ext` function that appends a `d11txhdr` structure to each frame before it is passed to the D11 core for transmission. To this end, frames are first enqueued with the `wlc_prec_enq` function and then transmitted by calling `wlc_send_q`. To change transmission parameters, we can place a hook at the end of the `wlc_d11hdrs_ext` function and change the `d11txhdr` structure accordingly.

*Our `sendframe` function accepts both frames with and without prepended `d11txhdr` which allows easy modifications of this struct.*

To inject arbitrary frames, Nexmon offers the `sendframe` helper function. It can send raw 802.11 frames starting with Wi-Fi headers. For those frames, `sendframe` calls the light-weight `wlc_sendctl` function discovered by Hoffmann in [41]. It takes raw frames, adds the `d11txhdr` structure, enqueues frames and triggers their transmission. Additionally, `sendframe` can handle frames that already contain the `d11txhdr` structure. Then `sendframe` only enqueues and sends those frames. The latter option is useful to gain more control over the transmission settings by manually calling the `wlc_d11hdrs_ext` function to create the `d11txhdr` structure and then modifying its contents before calling `sendframe`. In any case, frames for injection either need to come from the host or need to be crafted from scratch in the firmware. For the latter, we need to create an `sk_buff` structure by calling `pkt_buf_get_skb` and fill its data section with the raw frame bytes.

*The `pkt_buf_get_skb` function allocates new frames in the firmware.*



### 6.2.3 How are frames stored in the firmware?

In the firmware, each frame is stored in an `sk_buff` structure that contains a pointer to the frame itself (`data`), the frame length (`len`) and some flags (`flags`). The underlying data memory can be even larger and by calling the `skb_push` and `skb_pull` helper functions, we can move the data pointer forward and backward by automatically adjusting the `len` variable to prepend or remove headers. Generally, it is more efficient to first create a sufficiently large buffer that can hold all required headers and then shift the starting pointer to this buffer, instead of copying the whole frame payload to a new buffer whenever we are running out of space.

*The `skb_push` and `skb_pull` functions shift the beginning of a frame payload on the underlying buffer.*

### 6.2.4 How to handle retransmissions?

Retransmissions are handled by the D11 core. Whenever a transmitted frame requires an acknowledgment by the receiver, the frame is retransmitted as often as defined by the short retry limit (SRL) respectively the long retry limit (LRL). By default SRL is set to 6 and LRL to 7. We can change the values by using the `WLC_SET_SRL` and `WLC_SET_LRL` ioctls either with `nexutil` from userspace, or within the firmware by calling our `set_intioctl` helper function. For retransmissions, we can define up to four fallback rates on 802.11ac chips. The first is used for the first three retransmissions, the second for the fourth, the third for the fifth and the fourth for any other retransmission. To define those rates, we hooked the `wlc_antsel_antcfg_get` function that is called during the preparation of the `d11txhdr`. Using this hook, we get access to an instance of the `ratesel_txparams` structure that contains the `rspec` array to define the retransmission rates.

*SRL and LRL globally define the number of retransmissions for which we can define up to four different fallback rates.*

### 6.2.5 How to set transmit powers?

To override the transmit power of all outgoing frames, Broadcom offers the `qtxpower` iovar that can be set using the `WLC_SET_VAR` ioctl. In FullMAC firmwares, this setting can only choose transmit powers smaller than the regulatory limitations. To exceed these limitations, a debugging firmware is required that checks the `txpwroverride` variable that allows to override regulatory limits. As we also want to enable arbitrary power settings in production firmwares, we simply nop the call to the `ppr_compare_min` function that calculates the minimum between user targets and the regulatory limits. We need to place the nops into the `wlc_phy_txpower_recalc_target` function.

*By skipping over the selection of minimum powers set by a user and the regulatory limits, we can freely set any power index.*

The value set by `qtxpower` is first translated into a power index that the hardware uses to set actual gains at the amplifiers automatically. To also get full control over the amplifier values, we need to deactivate

*We can also manually define all amplifier values.*

hardware power control using the `wlc_phy_txpwrctrl_enable_acphy` function and can then abuse the `wlc_phy_txcal_cleanup_acphy` function to set all gains manually according to the definitions in the `ac_txgain_settings` structure.

### 6.2.6 What are the internal structures?

*We should never address structures with absolute addresses as their locations on the heap may change after patching the firmware binary.*

*The registers of the D11 core generally start at 0x18001000.*

To handle the internal state of the firmware, a number of structure instances are used and passed to functions. Most of these instances are created on the heap during the initialization of the firmware. Even though, they are always placed at the same positions in one firmware version, absolute references to these addresses should be avoided in the patch code as firmware patches allocating space on the heap can lead to address changes of these structures. If the location of one of the main structures is known, we can derive the addresses of the other structures. The `wlc_info` structure is the main structure handling the state of the high-layer driver functionalities such as the association state. It is mainly passed to functions starting with `wlc_`, but not to those starting with `wlc_bmac_`. The latter normally expect the `wlc_hw_info` structure managing hardware specific states such as access to the physical layer. Even more specific is the `phy_info` structure that controls the physical layer for each band and is passed to functions modifying amplifications or sending raw signals. The `d11regs` structure points to memory-mapped registers of the D11 core and gives direct control over its operation.

*Even FullMAC firmwares use operating system specific structures.*

The above mentioned structures are independent of the operating system. The `osl_info` structure keeps track of using operating system resources such as those used for the creation of `sk_buff` instances. Even though, no operating system is running on the Wi-Fi chip, Broadcom offers a minimal library with functions required to operate the Wi-Fi firmware. Another operating system specific structure is `wl_info` that is required by functions interacting with the operating system interface, for example, to pass frames from the firmware to the Linux kernel.

### 6.2.7 How to set channel specifications?

*Bypassing checks for valid channel specifications allows to select arbitrary channel numbers and bandwidths.*

For some experiments, researchers need to set restricted channel specifications (e.g., to use channel 14). On FullMAC chips, all available channels are defined in the firmware and only those allowed in the regulatory domain are selectable. These channels are also reflected in the operating system. Hence, by patching the firmware, we automatically modify the channels selectable by the host system. When the list of selectable channel specifications is generated at chip initialization or when changing regulatory domains, the `wlc_valid_chanspec_ext` function is called for all possible channel specifications. It returns

1 for every valid selection. To activate more channels, we hook the `wlc_valid_chanspec_ext` function and return 1 for any channel we intend to activate. This only allows to select channels that are standardized. To further set arbitrary specifications (e.g., to activate 80 MHz bandwidth in the 2.4 GHz band as used in Chapter 12), we need to patch the `wf_chspec_malformed` function to always return 0 to disable checking for a legal set of parameters.

*We can use 80 MHz bandwidth in the 2.4 GHz band.*

Generally, the transmission frequency is selected according to the chosen channel number. Nevertheless, we can select arbitrary carrier frequencies. For every channel a `chan_info` structure exists in an array. Every entry starts with the channel number, followed by the frequency and various phase-locked loop (PLL), bandwidth, mixer and amplifier settings. Varying these array entries for the currently selected channel results in frequency shifts observable on a spectrum analyzer.

*We can even choose arbitrary carrier frequencies.*

According to the regulations, channel 14 should only be operated in DSSS mode. To bypass this restriction, various firmware patches are required to figure out whether channel 14 was set or not. To simplify an unrestricted operation on this channel, we can overwrite the `chan_info` structure of another channel with the parameters of channel 14. Whenever this other channel is selected, the transmission is actually performed at the center frequency of channel 14 but without additional restrictions. Even though, setting arbitrary frequencies is possible, a licence is still required to operate a radio transmitter with the selected modulation at a chosen carrier frequency.

*Switching channel numbers in the `chan_info` array allows OFDM operation on channel 14.*

### 6.2.8 How to use timers?

Timers are an essential tool on microcontrollers to schedule tasks for periodic execution or execution at a later point in time. Whenever a timer times out, an interrupt is thrown and handled by an interrupt dispatches that executes callback functions defined in a timer structure. On Broadcom Wi-Fi chips, two different timers exist. The first one is part of the ARM core and counts in milliseconds. The second one is part of the D11 core and runs synchronous to the MAC layer. Its counts in microsecond steps. Even though, the second timer is more accurate, it is only available when the D11 core is active (minimum power consumption is deactivated). Hence, for regular scheduling tasks on the ARM core, we generally use the first timer.

*Broadcom chips have two timers accessible by the ARM core.*

To schedule the execution of a callback function with a given delay, the `hndrte_schedule_work` exists. It first initiates an `hndrte_timer` structure by calling `hndrte_init_timer` and then starts the delayed execution of the callback function by calling the `hndrte_add_timer` function. To start a periodic task, we added the `schedule_work` function that periodically reschedules the task until we delete the timer structure from the list of active timers by calling `hndrte_del_timer`.

*Various scheduler functions exist to execute a callback function with a delay or even periodically.*

*For each initialized timer structure separate callback function arguments are stored.*

Afterwards, we can free the assigned memory by calling the function `hndrte_free_timer`. To delay the first execution of a delayed task, we offer the `schedule_delayed_work` function. All of these scheduler functions take a pointer to a callback function that expects the `hndrte_timer` structure as first argument. Additionally, they take a pointer to a data variable or structure that is referenced within the timer structure and can be accessed by the callback function. This way, arguments can be passed to the callback function.

*The more exact timer executes a callback function whenever a timeout finishes.*

To initialize the second timer, the `wlc_hwtimer_allow_timeout` function allocates a timeout structure. After creation, we can pass it to the `wlc_hwtimer_add_timeout` function together with a delay in microseconds, a pointer to a callback function and a data pointer that is passed as first argument to the callback function when the timeout runs out. To periodically call a function, we have to call the add timeout function again in our callback function. To delete a timeout, we call the `wlc_hwtimer_del_timeout` function. Depending on the firmware version, `hwtimer` in the function names might be abbreviated to `hrt`.

#### 6.2.9 How to transmit arbitrary waveforms?

*We can either transmit raw signals from the sample-play buffer or from Template RAM.*

As described in Chapter 4, there are two approaches to transmit arbitrary waveforms. For transmitting mainly repeatable signals from a very short buffer (256 samples on a BCM43430 or 512 samples on a BCM4339), we store IQ samples in the sample-play buffer. On 802.11ac-capable chips, we can transmit longer waveforms, whose samples we store in the Template RAM. In both cases, we have to activate our transmitter and trigger the transmission. To make sure that the transmission path is active, we either have to send a Wi-Fi frame first or simply set a gain in the transmit amplifiers manually as described in Section 6.2.5.

*Every firmware contains functions to generate tones at a single frequencies for calibration purposes.*

The simplest signal we can transmit is a tone. To this end, we can call the `wlc_phy_tx_tone_acphy` function to generate the IQ samples at a given frequency, store the result in the sample-play buffer and start the playback. To generate and load IQ samples, this function calls the `wlc_phy_gen_load_samples` function and then it calls the `wlc_phy_runsamples_acphy` function to trigger the transmission. To be able to separate the signal generation and loading from the playback, we can call both functions individually. To generate the complex waveform, `wlc_phy_cordic` is called. For writing to the sample-play buffer, `wlc_phy_loadsampletable_acphy` is called. Those two function calls are also separable. For example, to generate a different waveform and load it into the sample-play buffer. The IQ samples are stored as 10 bit numbers combined in one 32 bit Integer `000000000000iiiiiiiiiiiqqqqqqqqq2`. During our experiments, we realized that some waveforms break the correct transmission. To

solve this problem, we need to deactivate clipping detection by calling the `wlc_phy_clip_det_acphy` function, which is called by the function `wlc_phy_stay_in_carriersearch_acphy`. The latter, however, makes the receiver deaf so that calling the it is not suitable for raw signal transmitters that need to react to incoming frames, such as a reactive jammer which we describe in Chapter 12.

To generate more complex signals than single tones, we offer various helper functions. In case we intend to create a long signal from the short signal stored in the sample-play buffer, we need to generate signals that have a periodicity that fits into the sample-play buffer. Using the inverse Fourier transform (IFFT) we can create exactly those signals. Multiples of the periods of each subcarrier fit exactly into the chosen number of samples that can even be smaller than the length of the sample-play buffer. To create these signals, we offer the `my_phy_tx_acphy_ext` function. While calling the function `wlc_phy_runsamples_acphy`, we can decide how often we want to play back the buffer or decide to continuously play it back, until we call `wlc_phy_stopplayback_acphy` to stop the continuous playback.

*Clip detection should be deactivated for arbitrary waveform transmissions.*

*The IFFT is the perfect function to generate continuously repeatable signals that fit into the sample-play buffer.*

#### 6.2.10 How to modulate information onto arbitrary waveforms?

The limited size of the sample-play buffer is a major downside of this signal transmission approach. As long as an 802.11ac-capable chip is used, we can bypass this limitation by using the Template RAM. However, the BCM43430 installed in Raspberry Pis cannot use this feature. To still transmit more advanced signals on this platform, we can use hardware components in the transmission path to modulate information onto a signal. In the simplest setup, we store a tone in the sample-play buffer and use it as carrier frequency for another signal. As illustrated in Figure 2 on page 18, a baseband multiplier can change the amplitude of the analog signal exiting the digital-to-analog converters (DACs). By continuously changing the value of the multiplication factor, we can perform an amplitude modulation of the carrier stored in the sample-play buffer. This allows us to transmit narrow band signals in the Wi-Fi bands. As both the inphase and quadrature components of the signal are modified in the same way, we end up with a double-sideband output signal. A quadrature modulation with a complex baseband signal could be possible as well by constantly modifying the inphase and quadrature calibration components to generate single-sideband signals. Preliminary experiments show, that modifying those calibration values has an effect on the image frequency rejection of the transmitted signals. Last but not least, we could try to modulate the carrier frequency directly by writing to the PLL controlling registers as described in Section 6.2.7. We leave the investigation of this approach as future work.

*Wi-Fi chips in non-802.11ac-capable hardware can transmit more advanced signals by modulating the analog baseband signal.*

*The baseband multiplier allows amplitude modulation, while calibration components might support quadrature modulation.*

6.2.11 *How to transmit raw signals from Template RAM?*

*On a BCM4358 we can store up to 3.2 ms of raw signal in the Template RAM at 40 MHz sampling rate.*

*We can decide for continuous playback or only transmitting the samples stored in Template RAM once.*

For playing back samples from Template RAM, the steps of creating samples, loading them and playing them back are very similar to using the sample-play buffer. However, we are not restricted to the small size of the sample-play buffer. Instead, we can use up to the full Template RAM that can hold 49152 samples on a BCM4339 or 131072 samples on a BCM4358. The IQ samples are stored as 32 bit Integers consisting of two 16 bit numbers for I and Q `iiiiiiiiiiiiiiiiqqqqqqqqqqqqqq2`. To write those samples into the Template RAM, we use `wlc_bmac_write_template_ram`. As the D11 core plays back those samples, we have to write the start and stop pointers of our signal into the `SamplePlayStart` and `StopPtr` registers, as mentioned in Section 4.2.3. We can access the two registers through the `d11regs` structure pointing at the D11 registers. To start the playback, we also call `wlc_phy_runsamples_acphy` but set the `mac_based` argument to 1, which prepares the transmit path to accept signals from the D11 core. Unfortunately, on the BCM4339 the latter function misses the instruction to trigger the transmission by the D11 core. To this end, we have to set some bits in the `psm_phy_hdr_param` register. We set bit `(1 << 1)` to enable the physical-layer clock, set bit `(1 << 11)` to enable signal playback and we can set bit `(1 << 12)` to end playback after transmitting the buffer once. In any case, we have to call `wlc_phy_stopplayback_acphy` after the transmission to stop the transmitter and go back to receiving signals. According to Jan Ruge, the `psm_phy_hdr_param` should be set to zero directly before setting the bits to transmit signals to increase stability.

6.2.12 *How to extract channel state information (CSI)?*

*Channel state information is stored in physical-layer tables.*

*To extract per-frame channel state information, an implementation in the D11 core is required.*

A widely requested feature in the research community is the extraction of channel state information (CSI) on Wi-Fi chips. While it is already possible on a selected number of Wi-Fi cards for notebooks and desktop computers, it was so far not available on smartphones. Extracting CSI on 802.11n/ac Broadcom chips requires to read from physical-layer tables containing this information. For each receive chain a separate table exists containing complex channel coefficients for all subcarriers and spatial streams send by the transmitter. In some firmwares or drivers a `wlc_phydump_chanest` function exists, that reads the contents of the CSI tables and dumps them as string. Unfortunately, it is not possible to dump CSI on a per-frame basis by using this approach. As soon as a new frame is received, the stored CSI is overwritten. Hence, we would need to extract the CSI during frame reception or make sure that the receiver is deaf while we extract the CSI. To this end, we cannot use the ARM processor as it only gets informed about received frames after their reception. Instead, we

have to implement the CSI extraction in the D11 core. In Chapter 8, we present an application that implements a CSI extractor able to extract CSI for each received frame. Nevertheless, after reading from the CSI table the payload of the transmitted frame becomes corrupted. Depending on the application, this downside may be negligible.

### 6.2.13 *How to talk to the firmware?*

For many applications, it is helpful to configure a firmware during runtime or extract information for debugging purposes. Below, we present means to directly access the chips memory (1), use the `printf` function (2), extract data through tunnels using the user datagram protocol (UDP) (3), use `ioctl`s to control the firmware (4) and send events from the firmware to the host (5).

To directly access the chip's internal memory (1), we can use `dhdutil` with its `membytes` option. It allows to read from and write to arbitrary memory locations in the RAM and may also directly read the ROM on some chips. Additionally, `dhdutil` offers the `consoledump` option that dumps the internal console buffer of the firmware to which we can write by calling the `printf` function (2). This allows to pass small amounts of textual data to the user space.

To send more data, we can encapsulate it in a UDP frame (3) and send it to the broadcast Internet protocol (IP) address 255.255.255.255. Those frames are always accepted by the Linux kernel and passed on into the user space, where they can even be received by apps without root privileges. To implement this in the firmware, we first create a new `sk_buff` buffer and fill it with the desired data and then prepend Ethernet, IP and UDP headers using our helper function `prepend_ethernet_ipv4_udp_header` (that uses UDP port 5500 by default). Then, we call the `xmit` function of the `wl` device to send the frame to the host.

Alternatively, to initiate transfers from the firmware, a user-space program such as `nexutil` can also initiate a synchronous data exchange with the firmware by calling `ioctl`s in the firmware (4). Each `ioctl` contains a command number, a pointer to a buffer to exchange data and the length of this buffer. `Ioctl`s can either only *set* data or set and *get* data back from the firmware. For the two directions set and get, `nexutil` offers the two parameters `-s<command_number>` and `-g<command_number>` and may either pass integers, strings, raw data from the standard input or base64 encoded raw data to the firmware. There, `ioctl`s are handled in the `wlc_ioctl` function that we hooked to check for custom `ioctl` command numbers and handle them in `ioctl.c`. To easily send back strings to the caller of a *get*-`ioctl`, we offer the `argprintf` function, that writes strings into the `ioctl` buffer and handles the remaining size automatically.

*Various ways exist to communicate with the firmware.*

*Dhdutil gives direct memory access and dumps the chip's console buffer on Android devices.*

*On any platform, we can send out information by encapsulating it in UDP datagrams.*

*Ioctl's allow for bi-directional data exchange initiated from the host.*

*Using argprintf, we directly print into an ioctl buffer.*

*Events originate in the firmware and are separately transferred from the data frames.*

*Events require handling in the driver.*

While `ioctl`s are always initiated by the host, the Wi-Fi firmware can also create an event (5) and send it as a message to the host, where we can handle it in the driver. The previously described option to send UDP datagrams can also be triggered by the firmware, but it creates additional frames which might interfere with regular experimental data. Hence, using the event messaging channel separates the control information from the data path. Nevertheless, handling events requires to recompile the driver or even the kernel, which is not always a suitable option. Daniel Wegemer figured out that we may create events by calling `wlc_event_alloc`, assign it to an interface using `wlc_event_if` and finally send it to the target interface by calling `wlc_event_process`. To handle frames in the `brcmfmac` driver (e.g., on the Raspberry Pi), we register new event handlers by calling `brcmf_fw_register` in the `brcmf_register_event_handlers` function. In our git repository [80], we offer examples for all five ways of communication as well as the sources to build firmware patches and the used utilities.

#### 6.2.14 How to modify the real-time firmware?

*The ARM firmware loads the real-time firmware during initialization.*

The real-time firmware is the ucode running in the programmable state machine (PSM) in the D11 core. In FullMAC chips, the ARM firmware contains the ucode as binary blob and loads it into the ucode memory of the D11 core. As only seven out of eight ucode bytes are actually used, some firmwares store the ucode with the eighth byte omitted. To extract those firmwares, we use our `ucodeext` utility. For ucodes that contain the eighth byte, we simply use `dd` to extract them from the ARM firmware.

*Using the b43-tools we can disassemble the ucode for analysis and modification.*

After extraction, we use the `b43-dasm` disassembler contained in the `b43-tools`<sup>2</sup> to disassemble the ucode. As illustrated in Figure 2 on page 18, the PSM has access to condition registers and special purpose registers (SPRs). To replace register numbers by speaking names defined in the `cond.inc` and `spr.inc`, we use the `b43-beautifier`. As it is still hard to understand the meaning of uncommented code, we intended to analyze it to figure out its meaning. To this end, Michael Koch found out in [51] that different ucodes have a very similar structure as the OpenFWWF firmware [33] created by Francesco Gringoli et al. for older BCM4306/11/18/20 Wi-Fi chips. Hence, by comparing code sections, we can get an understanding of how disassembled firmwares work.

To change the real-time behaviour of the firmware, we need to modify the Assembler code or use a tool such as the Wireless MAC Processor presented by Tinnirello et al. in [87] to graphically design state machines representing the behavior of the firmware. After modifying the ucode, we can reassemble it into a firmware binary and em-

<sup>2</sup> b43-tools repository: <https://github.com/mbuesch/b43-tools>



bed it after compression in the ARM firmware file. To avoid sharing the original ucode sources when publishing patches, we also provide means to apply patches containing only new code to freshly disassembled files. Overall, ucode modifications allow very advanced applications on off-the-shelf devices, such as partial packet recovery as presented by Han et al. in [35], or reactive jamming as presented in Chapter 12.

*Ucode modifications are required in advanced applications that influence Wi-Fi's real-time operation.*

#### 6.2.15 How to handle SoftMAC chips

Compared to FullMAC cards that implement the higher layer MAC operations in the ARM microcontroller of the Wi-Fi chip, SoftMAC cards implement those in the Wi-Fi driver running on the host's operating system. To modify the operation of those drivers, multiple options exist. If the driver source code is available (e.g., the `brcmsmac` driver or the `b43` driver), one can change it and rebuild the whole driver. If the driver is partially available as source code (e.g., for the `cfg80211` interface to the Linux kernel) and as object files (for device specific implementations), one can replace or hook original driver functions, by linking against object files that overwrite symbols of the original driver. This is a valid option to patch the proprietary `broadcom-wl` driver. If the driver is only available as binary (e.g., the `macOS` version of the `wl` driver), one may use the Nexmon approach to patch the driver as if it was a closed-source firmware running on a Wi-Fi chip.

*SoftMAC chips do not contain an ARM microcontroller to handle MAC operations in the firmware.*

### 6.3 DISCUSSION

Our Nexmon framework meets the challenges stated in the introduction. Using C as high-level programming language, we are able to write complex applications that can even be ported to different Wi-Fi chips. For debugging, we can print to the chip's console and by optionally using the debugging core that allows single-stepping, we are able to create detailed traces that allow for firmware analysis during runtime. Flashpatches enable changes of read-only memory and ucode compression frees space to place patches into firmware files. Further, we presented various ways to control the firmware during runtime and even extend it by side-loading code. Nevertheless, this chapter only scratches the surface of what is possible by reprogramming Wi-Fi firmwares. Especially in mobile wireless testbeds, Nexmon permits to implement algorithms in a battery saving manner with reaction times that were not achievable before. Because of its open-source nature, everyone can use the Nexmon framework to extend firmwares. We encourage researchers to also publish the source codes of their firmware extensions to enhance reproducibility of their work.

*The Nexmon framework meets the challenges of firmware patching.*

*Due to Nexmon, we can expect new wireless applications.*

## 6.4 CONCLUSION

*Nexmon realizes low-cost testbeds and it enhances reproducibility by using off-the-shelf devices.*

In this chapter, we introduced Nexmon—a tool to implement advanced applications in Wi-Fi firmware of FullMAC chips running on smartphones and IoT platforms. Due to Nexmon’s open availability, everyone can use our framework to setup their own testbeds. By supporting multiple wide-spread and low-cost platforms, we enhance the reproducibility of experiments and help to extend existing works to advance research. Unleashing the access to lower-layer capabilities through our patching framework allows to create new applications that run on off-the-shelf devices. This likely leads to a higher acceptability of results compared to SDR implementations.

## 6.5 MY CONTRIBUTION AND ACKNOWLEDGEMENTS

*While I developed the Nexmon framework, I was supported by various collaborators that analyzed firmware code and implemented applications for this platform.*

While Daniel Wegemer was working on analysing and patching the BCM4339 Wi-Fi firmware for his master thesis [90] using the tools provided by [10, 11], I came up with the idea of writing patches in C instead of Assembly and building a firmware patching framework that simplifies the creation of new patches. Since then, I started working on this framework. I thank Daniel Wegemer for accepting the challenge to start the research on Wi-Fi firmware reverse engineering and continuously supporting me in building, testing and promoting the Nexmon framework. In addition, I thank Francesco Gringoli for his intensive collaboration on analyzing and modifying the ucode running on the D11 core. I also thank Michael Koch for analysing the BCM4339 ucode in [51], Jakob Link for figuring out the meaning of CSI values [59], Justus Hoffmann for finding injection capabilities in [41], Andrés Blanco for supporting us in the early development steps, as well as Alexandr Potapov for his collaboration on reversing the Wi-Fi firmware.

To extend the functionality of a Wi-Fi firmware, we first have to reverse engineer the original firmware file. While finding functions by looking at strings in debug messages embedded in the firmware is quite easy, figuring out how a function implementation works in detail is more tedious. Especially, when no C code is available from which the binary under analysis was built. Static code analysis already gives a first impression of the function internals, but it does not consider the hardware's state during runtime. Some branches in the code might be unused on a certain platform. Additionally, the information stored in nested structure variables is complex to keep track of during static analysis.

*Static code analysis abilities are limited especially when the state of hardware components needs consideration.*

Hence, we saw the need for dynamic debugging approaches. The simplest approach is the creation of firmware patches that add debugging messages to a firmware. We can print them to the chip's internal console. To use those patches, we need to overwrite the code we want to monitor with a branch instruction that redirects the program flow into our debugging function. Then we need to execute the overwritten instructions and return to regular code execution. This approach works best by either overwriting the branch instruction that calls a function (BPatch or BLPatch) or the first instruction within the target function (HookPatch4). Overwriting arbitrary instructions is, however, tedious and error prone as the state of the registers, in particular the flag registers, and the state of the stack need to be considered to avoid program crashes or modifications of the states that lead to different paths through the program. Additionally, branch instructions are four bytes long while other instructions exist that are only two bytes long. Overwriting those requires to also overwrite surrounding instructions.

*To debug code, we can overwrite instructions to branch into our debugging functions.*

*Using this approach, we need to carefully handle the state of registers and the stack.*

Additionally, this patching approach is limited when code in ROM should be analyzed. To insert debugging hooks into ROM, we need flashpatches. They allow to overlay the ROM memory with up to eight bytes long blocks that should contain the branch instructions to the debugging code. The amount of flashpatches is, however, limited. Especially recent firmware versions that contain updated code sometimes have no free flashpatches left (e.g., firmware version 7.112.300.14 for the BCM4358) and some chips such as the BCM4335 do not even have a flashpatching unit.

*Analyzing ROM code requires flashpatches that are not available on every chip.*

Last but not least, replacing single instructions to branch to our debugging functions only results in limited debugging features. To really follow the program flow and analyze which branches were taken

*Implementing single-stepping debuggers in software is quite complex.*

during program execution, single stepping is required that calls the debugging handler after every instruction. While it is possible to implement such debuggers in software, they are quite complex as they have to consider instruction lengths and the types of instructions. For example, branch instructions can modify the program counter depending on conditions. To find the next instruction to patch requires to also evaluate the conditions manually.

*Dedicated debugger cores can halt the processor at every instruction.*

A solution to all the problems described above are dedicated hardware debugging cores. They can set a limited number of breakpoints without overwriting any instruction. Whenever the processors loads an instruction that has a breakpoint set, the debugging core is triggered and halts execution at this point. In most embedded development setups, the debugging core would completely stop the program execution on the processor and pass control over to an external debugger that is connected over JTAG. As the JTAG pins of the Wi-Fi chips in off-the-shelf devices are generally not accessible, we cannot use this setup in our case.

*Monitor-mode debuggers handle debugging events in exception handlers so that no external debugger is required.*

On devices that run complex operating systems with user interfaces, it is more common to not rely on external debuggers, but handle debugging events directly in software, which is called *monitor-mode* debugging. To this end, the debugging core generates exceptions whenever debugging events occur. Those exceptions also halt the execution of the currently running program but then redirect the execution to continue in an exception handler. This approach is similar to handling interrupt exceptions. The address of the exception handler is stored in the vector table at the beginning of each RAM firmware binary. A prefetch exception is triggered after hitting a breakpoint, while a data exception is triggered after hitting a memory watchpoint. The latter is a feature that can only be implemented using a dedicated debugging core. It triggers, whenever a program reads from or writes to a particular memory address.

*Only dedicated debugging cores allow to break on memory operations.*

*Handcrafting debuggers requires a thorough understanding of processor states.*

While developing the debugger, we faced the following challenges: (1) we had to find a way to access the debugging core registers without using a JTAG port, (2) we had to activate and unlock the debugging core, (3) we had to understand processor modes and exception handling on ARM microcontrollers, and (4) we had to reset a breakpoint after executing the instruction that triggered the breakpoint.

*We present challenges and implementation details.*

For this chapter, we implemented such a hardware-supported monitor-mode debugger for the ARM Cortex-R4 microcontrollers that execute the Wi-Fi firmware on Broadcom Wi-Fi chips. We first explain how to access the debugging core registers as well as the challenges we had to do so on Wi-Fi chips in Section 7.1. Then, we describe the implementation of our debugger in detail in Section 7.2, followed by an example debugging application in Section 7.3. Then, we discuss how our debugger helped us analyze Wi-Fi firmwares in Section 7.4 and conclude in Section 7.5.

## 7.1 ACCESSING DEBUGGING CORE REGISTERS

The ARM Cortex-R4's debugging registers are described in [21]. All debugging related registers are mapped into the microcontrollers memory. To figure out, where those registers are located, we have to execute a coprocessor instruction to read the Debug ROM Address register (DBGDRAR). To read from this register, we simply create a Nexmon firmware patch that uses the MRC instruction that moves the contents of a coprocessor register into an ARM core register from where we can dump it to the chip's console. On the BCM4339, the debug registers start at 0x18007000. We store this address as macro `DBGBASE` in the `debug.h` file in the Nexmon framework to avoid reading the coprocessor register during runtime.

*We need to read the base address of the memory-mapped debugging registers from a coprocessor register.*

The most important registers for setting breakpoints and watchpoints are the Breakpoint Control (DBGBCR), Breakpoint Value (DBGBVR), Watchpoint Control (DBGWCR) and Watchpoint Value (DBGWVR) registers. In the control registers, we set the conditions when a breakpoint or watchpoint should be triggered. In the value registers, we set the address on which we want to trigger. To activate debugging itself and set it to monitor-mode debugging, we need the Debug Status and Control register (DBGDSCR). Last but not least, accidental access to the debugging registers should be avoided. Hence, before accessing those registers, we need to write a magic number (0xC5ACCE55) into the Lock Access register (DBGLAR).

*Only a handful of debugging core registers is required to setup breakpoints and watchpoints.*

During our first attempts to read from the memory mapped debugging registers, we realized that it is possible to read from them at the beginning of the initialization phase. Whenever, we tried reading them afterwards, the chip crashed. By trying to access the debugging registers at various instructions, we narrowed down a call to `si_update_chipcontrol_shm` in the `si_setup_cores` function. Before this call, accessing the debugging registers works, while afterwards it crashes the chip. Hence, we inferred from this observation that this write to the chip control shared memory disables the debug core in the ARM chip—likely to save energy. By removing the call to `si_update_chipcontrol_shm` we can keep the debugging core accessible in any firmware state. In the next section, we explain how we implemented our programmable debugger with the Nexmon framework.

*During regular firmware operation, the debugging core gets disabled in the initialization phase.*

## 7.2 IMPLEMENTATION

As all of our projects, we also published the source code of our debugger application to make it reusable by the community (see Section A.3). We use `debugger_base.c` to store initialization functions and exceptions handlers required for every debugger implementation. We use `debugger.c` to set breakpoints and watchpoints and

*Our example debugger application is publicly available for download and experimentation.*

implement the corresponding handlers. The `debug.h` header file is included in the Nexmon framework to make it reusable by various Nexmon based projects.

### 7.2.1 Initializing the debugger

*During an abort exception, the processor changes its mode and uses banked stack pointer and link registers.*

*We need to reserve space in RAM for the abort mode's stack.*

*Regular breakpoints and watchpoints need to be set to trigger on an address match.*

To initialize the debugger, we hook the call to the `c_main` function (which is the first C function called in the firmware) and prepend a call to `set_debug_registers`. There, we first assign a memory region for the stack in the processor's abort mode. The microcontroller can run in various modes that have different privileges and are also used as conditions to avoid triggering breakpoints and watchpoints. Generally, the firmware is executed in system mode but after a prefetch abort or data abort exception occurs, the processor automatically changes into abort mode. The original firmware directly changes back into system mode to print a register and stack trace, but we want to stay in abort mode as it gives us a separate stack as well as link and stack pointer registers to run our debugging code. By doing this, we do not influence the state of the original stack by calling functions in our debugging handler. Hence, we need to reserve some space in the RAM for the abort-mode stack.

After adjusting the abort-mode stack, we unlock access to the debugging registers, disable all breakpoints and activate monitor-mode debugging. Then we define our breakpoints and watchpoints to trigger on an address match. That means, whenever the program counter reaches a breakpoint address a prefetch-abort exception is triggered and whenever a load or store instruction reads from or writes to a watchpoint address a data-abort exception is triggered.

### 7.2.2 Preparing to handle breakpoints and watchpoints

*We need to override exception handlers to stay in abort mode.*

*After handling breakpoints and watchpoints, we may continue execution of the original firmware.*

To handle prefetch-abort and data-abort exceptions, we first override the original handler functions `tr_pref_abort` and `tr_data_abort` as they directly switch the processor mode back to system mode. In our implementation, we stay in abort mode. Then we call the function `handle_exceptions` that is called whenever an exception occurs. It pushes all registers to the stack so that we can analyze them in our handler functions. Then, it calls the `choose_exception_handler` that dispatches the exception handlers. After returning to the function `handle_exceptions`, the pushed registers are popped from the abort-mode stack and the processor changes back to system mode, where it continues to execute the firmware. This implementation allows us to handle breakpoints and watchpoints and then continue with the regular firmware operation.

Within the `choose_exception_handler` function, we either call the function `handle_pref_abort_exception` to handle prefetch-abort ex-

ceptions or the function `handle_data_abort_exception` to handle data-abort exceptions. In both handlers, we first need to fix the values for the link and stack pointer registers previously pushed to the stack as they are set to the register values of the abort mode. For handling debugging events, we are more interested in the link and stack pointer registers of the system mode. To get those values, we call the `fix_sp_lr` function. It disables monitor-mode debugging, switches back into system mode, copies the values of the link and stack pointer registers into register R1 and R2, switches back to abort mode and activates monitor-mode debugging again. Then, we override the link and stack pointer registers on the stack with the values extracted through registers R1 and R2. The address of the pushed register values on the stack is passed as first function argument to our handler functions and interpreted as a trace structure that contains the values of all system-mode registers. Using this structure, we cannot only read the register contents, but also modify them, as the `handle_exceptions` function pops them from the stack before continuing regular firmware execution. In the following two subsections, we describe the handling of breakpoints and watchpoints in more detail.

*At the beginning of our handler functions, we push the system mode link and stack pointer registers.*

*We interpret the pushed registers as trace structure in which we may modify register contents before continuing execution.*

### 7.2.3 Handling breakpoints

We handle breakpoints in the `handle_pref_abort_exception` function. To this end, we first check which of the four possible hardware breakpoints is enabled. For each enabled breakpoint, we first check if the program counter address in the trace structure equals the address stored in the corresponding Breakpoint Value register. Now, we may handle the breakpoint by checking or modifying register values or writing debugging information to the console. Then, we either have to disable the breakpoint to continue execution, or we have to change it from an address match breakpoint into an address mismatch breakpoint. Otherwise, we would end up in a continuous breakpoint loop, as everytime we want to execute the instruction our breakpoint triggers on, the breakpoint would be triggered again. By triggering on an address mismatch, we can execute exactly this instruction and then trigger another breakpoint on the next instruction that should be executed. To know, to which breakpoint the next prefetch-abort exception belongs, we mark the hardware breakpoint number in a variable.

*If the current system-mode program counter equals the address of an active breakpoint, we can execute our breakpoint handling code.*

If the program counter does not match the breakpoint address, we check if this variable marked the breakpoint and know that we now have to handle the address mismatch breakpoint. In the simplest case, we would reset the breakpoint to trigger on an address match and continue execution to wait until the breakpoint triggers again. As for handling the address match breakpoint, we can also output some debug information. Alternatively, we could decide to set the

*We implement single-stepping by resetting address mismatch breakpoints to the current program counter address.*

*Our programmable debugger allows creative debugging implementations.*

breakpoint again to trigger on an address mismatch at the current program counter location. This allows us to implement a single-stepping debugger that triggers on every instruction that should be executed. Using it, we can programmatically analyze which paths are taken through a program. To exit single-stepping mode, we can either disable the breakpoint or set it to trigger on the original breakpoint address again. As exit conditions can be programmed, we can implement complex debugging scenarios. For example, we could exit after a defined number of steps, or as soon as the return instruction is executed, or as soon as a loop counter reaches a defined value. This renders our debugger implementation very flexible.

#### 7.2.4 Handling watchpoints

*Figuring out which watchpoint triggered an exception is more complicated than for breakpoints.*

Handling watchpoints is a bit more complicated than handling breakpoints. Unfortunately, there is no register pointing at the address that triggered the watchpoint. To find out which of the four watchpoints was set, we would have to analyze the current instruction and extract which register holds the memory address for the current load or store instructions. For simplicity, we do not differentiate between different watchpoints and always assume that the first watchpoint was triggered in our `handle_data_abort_exception` function. Then we disable all watchpoints so that we are able to execute the load or store instruction without triggering the data-abort exception over and over. If we intend to reactivate the watchpoint, we have to create a breakpoint mismatching the current program counter address so that it triggers after the execution of the load or store instruction. For this operation, we reserved the fourth breakpoint. In the `handle_pref_abort_exception` function, we handle this breakpoint and enable the watchpoint again. As for breakpoints, we can also modify memory locations and registers and print debug information to the console when we handle watchpoints. In the following section, we demonstrate the debugger abilities in an example.

*To reenale a watchpoint, we have to trigger a breakpoint mismatching the current program counter address.*

### 7.3 EXAMPLE APPLICATION

*In our example application, we set a breakpoint on the printf function.*

In our example application, we demonstrate the ability to set and handle breakpoints and watchpoints, and perform single-step debugging. We set our breakpoint at the first instruction of the `printf` function. In the debugging handler `handle_pref_abort_exception`, we check if the `printf` function's second argument equals the string `sdpcmd_dpc` and only then print a debug message. To print to the console, we again use the `printf` function that has a breakpoint set. To avoid triggering the breakpoint again, we make sure that the breakpoint triggers in system mode but not in abort mode. This allows us to set breakpoints on any function in the firmware without having to care



if it is used somewhere within the functions called by the debugging handler. Nevertheless, we need to take care that handler functions do not scramble hardware registers required by the currently debugged code. For example, we could generate events in our debugger and handle them in the hosts driver as described in Section 6.2.13. The transmission of these event messages, however, uses the DMA controllers between Wi-Fi chip and host, so that we need to make sure that the function we are debugging is currently not using those registers. By simply calling the `printf` function, we are generally on the safe side as long as it only prints to an internal console buffer.

Besides printing a debug message, we also activate single stepping only when the second `printf` function argument matches our target string. Then we retrigger the breakpoint with address mismatches again and again, until our step counter reaches four. Then we rearm the breakpoint to trigger on the beginning of the `printf` function. In each step, we print the current program counter address.

Regarding our watchpoint, we set it to trigger on the address of the string `%s: Broadcom SDPCMD CDC driver` that gets printed at the beginning of the initialization phase. After triggering the watchpoint once, we rearm it up to 20 times using an address mismatch breakpoint as described above. On every data-abort exception, we print the address of the current program pointer.

In Listing 1, we illustrate the output of our debugger example application. One can observe that the watchpoint `WP0` was triggered three times. The first two are load instructions in the `vsprintf` function called by `printf`. The third is a store instruction within the `memset` function. As the string is only needed during initialization, it is stored in a memory section that is first overwritten and then assigned to the heap in the `hdrte_reclaim` function, whose output is printed directly after our watchpoint output. Then we see the output of the first breakpoint `BP0`. In step 0, the breakpoint is triggered on the first instruction of the `printf` function. In the following steps, the breakpoint retriggers on address mismatches and the program counter `pc` increases.

## 7.4 DISCUSSION

We mainly developed the debugger to analyze code during runtime and understand how it works to build our patches. It helped us tremendously to analyze which function is best to inject frames and which arguments need to be passed at the function call to avoid crashing the firmware. Before we discovered flashpatches, it was the only way to debug code stored in ROM. Unfortunately, we realized during our experiments, that setting breakpoints makes the firmware unstable, when it is using interrupts to react to external events. Still, the debugger works sufficiently well to, for example, set a breakpoint, out-

*We can even call functions that include a breakpoint from our debugging handler without triggering the breakpoint again.*

*We activate single-stepping to print the program counter of the next four instructions.*

*We trigger based on a string comparison.*

*The watchpoint triggers on load instructions in the function `vsprintf` and on a store instruction in the function `memset`.*

*The debugger helped us understand the injection of raw Wi-Fi frames.*

*The debugger is also reusable on other ARM Cortex-R4 platforms that require debugging but do not have an accessible JTAG port.*

put some debug messages during single-step debugging, dump the console buffer for analysis and then simply reload the firmware for the next debugging experiment. Overall, our implementation meets the challenges described in the introduction. Last but not least, the debugger implementation is not limited to Broadcom Wi-Fi chips, it should be reusable to debug any ARM Cortex-R4 microcontroller in embedded devices, even if the JTAG port is not accessible.

## 7.5 CONCLUSION

*Our debugger implementation solely relies on default ARM microcontroller capabilities which makes it portable to other embedded systems.*

In this chapter, we presented our integrated monitor-mode debugger for Broadcom Wi-Fi chips. It accesses the debugging core of ARM Cortex-R4 processors to set and handle hardware breakpoints and memory watchpoints. To this end it uses the monitor debugging mode that triggers an exception in the firmware to call debugging handling functions. This way, we do not rely on a JTAG port to dynamically debug the firmware running on a Wi-Fi chip. As our implementation directly accesses features of ARM microcontrollers it can also be ported to other systems where embedded firmwares need to be analyzed but the JTAG port is not available. Additionally, it allows to quickly react to debugging events and continue execution

Listing 1: Output of our debugger example application.

```
RTE (USB-SDIO-CDC) 6.37.32.RC23.34.43 (r639704) on BCM4339 r1 @
37.4/161.3/161.3MHz
000000.010 WP hit pc=00012b3a
000000.013 WP hit pc=00012b3a
000000.010 sdpcmdcdc0: Broadcom SDPCMD CDC driver
000000.141 reclaim section 0: Returned 31688 bytes to the heap
000000.189 nexmon_ver: 63fb-dirty-14
000000.192 wl_nd_ra_filter_init: Enter..
000000.196 TCAM: 256 used: 198 exceed:0
000000.200 WP hit pc=000126c2
000000.203 reclaim section 1: Returned 71844 bytes to the heap
000000.208 BP0 step 0: pc=000126f0 *r1=sdpcmd_dpc
000000.213 BP0 step 1: pc=000126f2
000000.216 BP0 step 2: pc=000126f4
000000.219 BP0 step 3: pc=000126f6
000000.223 BP0 step 4: pc=000126fa
000000.226 BP0 single-stepping done
000000.229 sdpcmd_dpc: Enable
000000.234 wl0: wlc_bmac_ucodemibss_hwcap: Insuff mem for MBSS:
templ memblks 192 fifo memblks 259
000000.249 wl0: wlc_enable_probe_req: state down, deferring
setting of host flags
000000.295 wl0: wlc_enable_probe_req: state down, deferring
setting of host flags
```

as soon as possible without the overhead of communicating with an external debugger. In the next chapter we introduce our channel state information extractor. It is a tool to dump advanced physical-layer information on per-frame basis.

## 7.6 MY CONTRIBUTION

When we had the need to dynamically analyze firmwares and the ROM was not yet patchable, I came up with the idea of directly accessing the ARM debug core of the Wi-Fi chip. I performed experiments to understand when and how to access the debug registers and implemented the debugger application in the form it currently has.

*There was a need for dynamic debugging capabilities.*



## CHANNEL STATE INFORMATION EXTRACTOR

During regular reception, every OFDM-based Wi-Fi receiver needs to first extract channel state information (CSI) from the long-term training field (LTF) of a frame's preamble to cancel the effects of the wireless channel and to demodulate the transmitted data. In the last few years, various applications based on extracted channel state information emerged. Most of them are based on the Intel CSI tool created by Halperin et al. and presented in [34]. It allows to extract CSI on Intel Wi-Fi chips installed in PCI-Express extension cards for laptops and desktop computers. Though, their tool delivers the CSI on a per-frame basis, laptops and desktop computers do not have the mobility and omnipresence of smartphones. Hence, we decided to develop a CSI Extractor application for BCM4339 Wi-Fi chips installed in Nexus 5 smartphones. During the development, we faced the following challenges: (1) We had to find and access the memory that holds the CSI, (2) we had to extract the CSI before the next incoming frame overwrites the CSI memory, and (3) we had to interpret the format of the stored numbers correctly.

By crawling through Broadcom driver source codes as well as the `wl` driver running on macOS, we found the `wlc_phydump_chanest` function. It is a debugging function, that reads and dumps values from a physical-layer table (see Figure 3 on page 21). Unfortunately, the function is implemented in the ARM firmware and a call to the function does not make sure that no other frame reception corrupted the stored CSI between receiving the last frame and calling the CSI dumping function. Additionally, this function is not available in the production releases of FullMAC firmwares such as the one for the BCM4339 chip used in smartphones. At least we learned in which memory the CSI can be found and how to extract it. In Section 8.1, we describe how we use this information to extract CSI using the real-time processor (D11 core) of the Wi-Fi chip. In Section 8.2 we use a simple example setup to evaluate CSI dumps from two smartphones communicating with each other. In Section 8.3 we discuss our results and conclude with related work that might benefit from our implementation in Section 8.4, followed by our conclusion in Section 8.5.

## 8.1 IMPLEMENTATION

The CSI contains the amplitude and phase changes on each subcarrier introduced by the wireless channel. As this information is required to equalize the channel's effect on the transmitted symbols, the CSI

*Channel state information extraction on commercial devices already lead to various new applications.*

*Up to now, no CSI extraction platform existed for smartphones.*

*The `wlc_phydump_chanest` function revealed the general ability to dump channel state information on Broadcom chips.*

*To gain per-frame CSI we need an implementation in the D11 core.*

*The CSI is stored in a physical layer table on Broadcom chips.*

is available in all OFDM-based Wi-Fi receivers. On Broadcom and Cypress cards, the physical layer extracts this information and stores it in a physical-layer table. As the content of this table changes for every received frame, we have to read it during frame reception. The ARM microcontroller running the FullMAC firmware is not suitable for this task, as it processes only complete frames. Hence, we need to modify the code running on the D11 core's programmable state machine.

### 8.1.1 The size of channel state information

*The CSI contains complex numbers for each spatial stream, each receive antenna and each subcarrier.*

*The CSI of a Wi-Fi frame can easily exceed the length of the frame's payload.*

*On Nexus 5 smartphones, the maximum CSI size is limited to 968 bytes per frame.*

*Transferring CSI from the D11 core to the ARM core bears another challenge.*

First, we analyze the maximum size of the CSI information data structure. The CSI itself consists of complex numbers  $H_{sts,rx,sc}$  indicating phase and amplitude changes for each of the transmitted spatial streams  $sts$ , each receive antenna  $rx$  and each subcarrier  $sc$ . As each of the complex numbers is stored in 32-bit values, the total number of CSI bits per frame equals  $32 \cdot sts \cdot rx \cdot sc$ , respectively,  $4 \cdot sts \cdot rx \cdot sc$  bytes. Each 802.11ac node can have between one and eight antennas, defining the range for  $sts$  and  $rx$ . The number of subcarriers depends on the used channel bandwidth and equals  $\{64, 128, 256, 512\}$  for  $\{20, 40, 80, 160\}$  MHz bandwidth. Only considering non-zero subcarriers, these numbers reduce to  $\{56, 114, 242, 484\}$  used subcarriers. Hence, the smallest CSI array consists of  $4 \cdot 1 \cdot 1 \cdot 56 = 224$  bytes, while the largest consists of  $4 \cdot 8 \cdot 8 \cdot 484 = 121$  KiB.

For our implementation, we use Nexus 5 smartphones. They feature a single Wi-Fi antenna and support 80 MHz bandwidth. Hence, the largest CSI consists of  $4 \cdot 1 \cdot 1 \cdot 242 = 968$  bytes, which is roughly 69 times larger than a Wi-Fi acknowledgement frame (14 bytes including FCS bytes). A key challenge for our system is to extract this amount of control information for every received frame. For the extraction, the D11 core needs to first copy the CSI from the physical-layer CSI table into the D11 core's shared memory, which is limited in size and also stores variables that are required for regular Wi-Fi operation that should not be overwritten. After copying the information, we face the problem of transferring it to the RAM of the ARM core. Even though the ARM core can directly read from the shared memory, it is not guaranteed that the ARM core handles a frame reception sufficiently quick before the CSI in the shared memory is overwritten by the next incoming frame.

### 8.1.2 Pushing channel state information out of the D11 core

The optimal way for transferring CSI from shared memory to the ARM core's RAM would be to trigger a DMA transfer on the whole CSI containing memory region. To the best of our knowledge, the DMA controllers do not support this operation. Nevertheless, the

D11 core prepends an additional header containing meta information to each received frame. This header is normally 28 bytes long and read from the shared memory during DMA transfers that copy the frame's payload into the ARM core's RAM. We realized that multiple of these DMA transfers can be triggered in a row. In this case, the transferred frames contain an empty payload, but the shared memory contents are copied in each transfer. By adjusting the meta header's start pointer after each transfer, we can copy the complete CSI from shared memory to the ARM core's RAM. Additionally, we managed to increase the size of those transfers from 28 to 64 bytes. Using four bytes for a header, 17 transfers are required to copy the whole 80 MHz CSI information in single antenna systems. In the ARM core, we detect CSI transfers according to this header and can either process the received CSI directly or send it to the host using UDP frames. To make this implementation reusable by the community, we published its source code as described in Section A.6. In the following section, we demonstrate what can be achieved with out implementation in practice.

*The most efficient way to transfer information from D11 core to ARM core is using a DMA controller.*

*We need 17 DMA transfers to copy the whole CSI for one transmit and one receive antenna.*

## 8.2 EXPERIMENTAL EVALUATION

As environment we choose an apartment in a rural area with only low amount of Wi-Fi traffic so that our experiments are not affected by high levels of interference. For influencing the propagation characteristics of the wireless channel in a repeatable fashion, we placed one device into a microwave oven and continuously changed the opening angle of the microwave door. We illustrate our setup in Figure 5.

*We perform our experiments in an apartment with low Wi-Fi traffic.*

### 8.2.1 Experimental setup

Extracting channel state information on off-the-shelf devices can lead to various interesting applications. As evaluating our CSI extractor against all existing solutions is out of scope of this work, we consider a simple example applications that demonstrates the performance of our extractor. To this end, we setup two Nexus 5 smartphones in the kitchen of an apartment. One goes into our microwave oven, the other one on the kitchen countertop facing the microwave oven as illustrated in Figure 5. Even with a closed microwave-oven door, the two nodes can exchange Wi-Fi frames on the 80 MHz wide channel 122 in the 5 GHz band. Nevertheless, the position of the oven door heavily influences the propagation characteristics of the wireless channel between our two smartphones. We intend to illustrate this effect, by extracting the channel state information on both of the two nodes.

*Our setup consists of two Nexus 5 smartphones, one in a microwave oven the other one on a kitchen countertop.*

*We open and close the microwave oven door to influence the propagation characteristics.*

We configure the node outside the microwave oven to transmit frames every 100 ms directly from the firmware. The node in the mi-

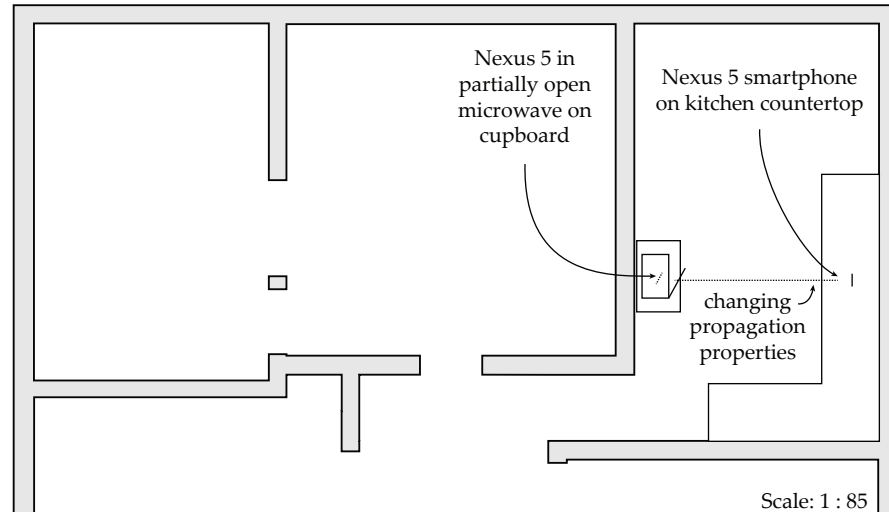


Figure 5: Experimental setup in an apartment in a rural environment with no other Wi-Fi traffic on channel 122 and 120. Placing the transmitter in the partially open microwave increases multi-path effects. All smartphones are installed on car mount holders to enhance the antenna radiation characteristics. (based on [75])

*CSI is extracted by the D11 core, forwarded to the ARM core, packed into UDP datagrams and send to the host.*

crowave oven receives those frames, extracts the CSI in the D11 core and sends it up to the ARM core. After the CSI is fully received, the ARM core stores it in a UDP datagram and forwards it to the user space, where we dump it using tcpdump. Additionally, the ARM core crafts a new frame and sends it back to the other smartphone, which also extracts the CSI and forwards it to user space for dumping. While the two nodes exchange frames and dump CSI values, we open and close the microwave oven door three times. The first two times rather slowly, the third time a bit faster.

### 8.2.2 Analyzing the CSI dumps

*We analyze the CSI dumps in MATLAB and plot their magnitudes as waterfall diagram.*

*The results illustrate the reciprocity of the wireless channel and the repeatable effect of the door's position on the channel.*

After running the experiments, we collect the frame dumps from the two smartphones. Then, we analyze the per frame CSI information in MATLAB and plot the magnitudes of the CSI for every subcarrier. We illustrate the result in Figure 6. At the top, we plot the CSI of the first exchanged frame for both of the two nodes. Below, we plot the magnitudes of the following CSI frames as a waterfall diagram. Comparing the two waterfall diagrams as well as the single plots clearly shows the similarity of the CSI measurements in both directions. This lets us imply the reciprocity of the wireless channel, which is an expected result. Additionally, we can observe repeating pattern in the waterfall diagrams. They repeat whenever we repeat the action of opening and closing the oven door. Regarding both actions, the pattern during closing the door is mirrored into the pattern of opening the door. Which indicates a very stable wireless environ-



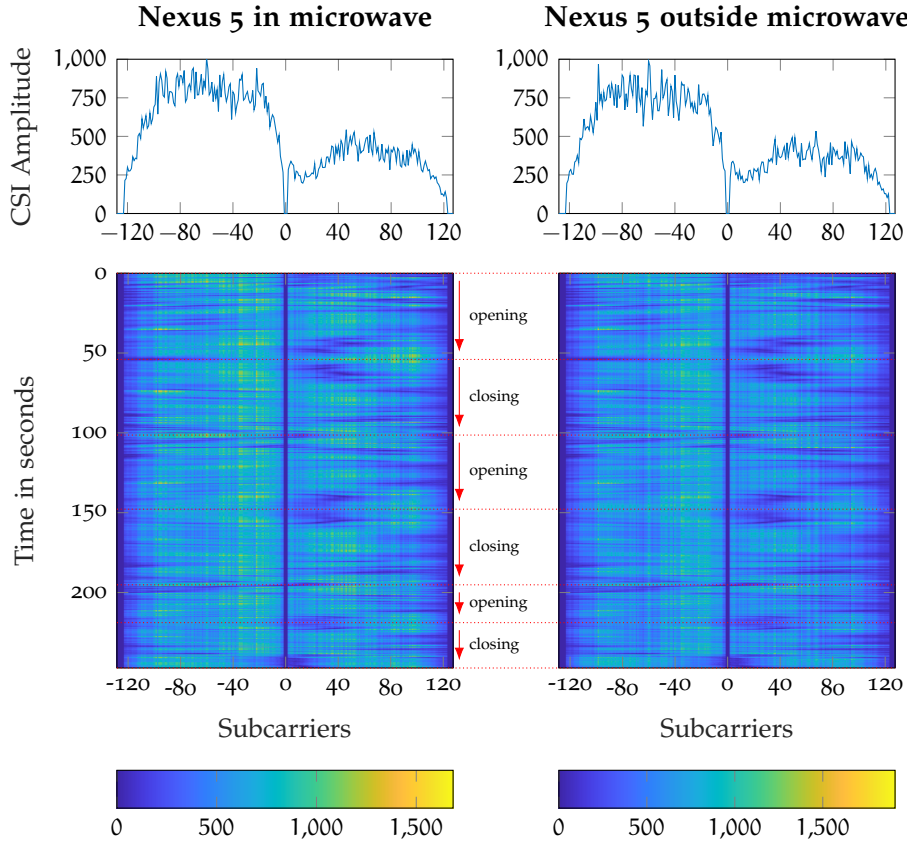


Figure 6: Continuous bi-directional channel state information (CSI) measurement between two Nexus 5 smartphones, one in a microwave oven, the other one outside, while opening and closing the microwave oven door three times. The measurement is taken at channel 122 with 80 MHz bandwidth in the 5 GHz band. At the top, we illustrate the amplitudes of one CSI measurement, followed by a waterfall diagram with 10 measurements per second. The channel reciprocity is clearly observable. (based on [75])

ment, in which the wireless channel between the two smartphones is mainly influenced by the position of the oven door.

### 8.3 DISCUSSION

The ability to dump uncompressed channel state information on a per-frame basis with smartphones allows to either enhance already existing solutions by adding mobility or even implement new applications. Compared to already available CSI extraction firmwares for other off-the-shelf Wi-Fi chips, we support 802.11ac frames with currently 80 MHz bandwidth, which gives at least twice as much insight into wireless propagation characteristics as existing solutions offer. The open source nature of our firmware patch allows modifications by other researchers. This results in the flexibility of using the Wi-Fi

*Our solution is the first that supports 802.11ac chips.*

*The open source nature allows for easy extendability by the community.*

chip as coprocessor that preprocesses CSI directly in the Wi-Fi chips ARM processor and only delivers aggregated information to an app running on the smartphone. The similarity of all Broadcom Wi-Fi chips also allows to port our solutions to other devices that have more antennas and offer even wider bandwidth of up to 160 MHz.

#### 8.4 RELATED WORK

*CSI extraction tools already exist for Intel, Atheros and Broadcom cards but none of them run on 802.11ac chips that support 80 MHz bandwidth and none of them support Wi-Fi chips in smartphones.*

While our CSI extractor is the first that runs on smartphones and supports 802.11ac transmissions, two other solutions existed for 802.11n devices. The first was developed by Halperin et al. for Intel Wi-Fi cards and presented in [34]. Unfortunately, the firmware patch<sup>1</sup> is closed source so that it is hard to port it to newer Intel Wi-Fi cards or circumvent its limitations of only extracting every second CSI value on 20 MHz channels or every fourth on 40 MHz channels. For Atheros cards, Xie, Li, and Li developed a tool<sup>2</sup> that they use in [94] to produce precise power delay profiles. It can extract CSI at all subcarriers and is open source to support various Atheros chips. Ricciato et al. developed another CSI extractor that is close to our implementation, but it only supports 802.11n Broadcom SoftMAC chips that neither support 80 MHz bandwidth or operation in smartphones. They use their tool in [72] to enhance position and velocity measurements based on packet arrival times by using the extracted CSI.

*One can detect device tampering using CSI.*

Besides the tools themselves and the works these tools were first used in, various other authors developed applications that require channel state information. In [4], Bagci et al. use CSI to detect whether someone tampered with a device. For example, by moving or replacing a wireless surveillance camera. To cope with dynamic environments with some movements, Bagci et al. take multiple CSI extraction nodes into consideration. In [47], Jiang et al. use CSI to detect spoofing attacks in Wi-Fi networks. While data frames are encrypted, management frames are vulnerable to spoofing which allows denial-of-service attacks by injecting deauthentication frames. As Jiang et al. show, using CSI to detect spoofed frames is eight times more reliable than using received-signal-strength measurements. In [1], Ali et al. use CSI information to reconstruct keystrokes. Using Nexmon, one could also try to extract user inputs from a smartphone. Besides [72] (mentioned above), numerous localization schemes exist that use CSI. In [56], Li et al. do not even require that a tracked person carries a transmitting device. It is sufficient that people pass between a frame transmitter and multiple receivers that estimate the CSI. Then, Li et al. apply their Doppler-MUSIC and Doppler-AoA algorithms to estimate the velocity and the location of a target. In [26], Fang et al. use CSI to measure human activity and vital signs. To this end, they use a

*One can detect spoofing attacks using CSI.*

*One can detect keystrokes using CSI.*

*One can even track persons that do not carry devices themselves using CSI.*

<sup>1</sup> Intel CSI tool: <https://dhalperi.github.io/linux-80211n-csitol/>

<sup>2</sup> Atheros CSI tool: <http://pdcc.ntu.edu.sg/wands/Atheros/>

wristband to transmit Wi-Fi frames that are captured by the Intel CSI tool attached to a small single-board computer that is worn by the test person. Using our CSI extractor instead, one could replace the single-board computer by a smartphone that people regularly carry anyways. The presented work only gives a rough overview of possible CSI based applications that can likely be improved by using smartphones for CSI measurements.

*One can monitor vital signs and human activity using CSI.*

## 8.5 CONCLUSION

In this chapter, we introduced our Nexmon CSI Extractor. It is the first tool that allows dumping channel state information on Broadcom FullMAC chips on a per-frame basis. We use the D11 core to first copy the CSI from physical-layer tables into the shared memory and then trigger DMA transfers to copy the information into the ARM core's RAM. From there, we forward the CSI to the host encapsulated in UDP datagrams. This potentially gives access to channel state information to any application running on a smartphone and eases the development of new applications that require CSI. In Chapter 13, we use our CSI Extractor to implement a covert channel that hides information by prefiltering outgoing Wi-Fi frames. In the following chapter, we present another novel tool for Wi-Fi chips, that turns off-the-shelf devices into software-defined radios.

*Our tool is the first to both dump channel state information on smartphones and to work with 802.11ac hardware.*

## 8.6 MY CONTRIBUTION AND ACKNOWLEDGEMENTS

Especially for the covert channel application presented in Chapter 13, I saw the need for extracting channel state information on smartphones. I thank Francesco Gringoli for helping me with the D11 firmware implementation, especially for providing the idea to transfer CSI information from the shared memory to the ARM core by triggering multiple DMA transfers in a sophisticated way, as also used in [72]. After implementing the extraction mechanism, we had to interpret the CSI values correctly. For the latter, I thank Jakob Link, who figured out how real and imaginary parts of CSI dumps are encoded in his bachelor thesis [59].

*Extracting and interpreting CSI was a joint work.*



Wi-Fi can be regarded as the de-facto standard for wireless local area networking, and the installed base is in the billions. Adhering to the Wi-Fi standard provides for interoperability and serves the basic communication needs such as Internet access. Most Wi-Fi chips integrate more advanced features, which are usually neither documented nor exposed to developers or end users. For example, we are able to demonstrate how to turn off-the-shelf devices, such as smartphones, into full-fledged software-defined radios (SDRs) by utilizing the aforementioned undocumented features. This provides multiple benefits over existing SDR platforms such as WARPs or USRPs. The latter are built with full flexibility in mind, but do not run regular software stacks and applications easily. Instead, converting off-the-shelf devices into SDRs allows to easily scale up experiments to hundreds of SDR-enabled nodes, while retaining the possibility to run regular mobile apps and supporting device mobility. This also facilitates experimentation outside of lab environments. Furthermore, the device specific transmit and receive characteristics are maintained.

To transmit arbitrary waveforms, we have to store IQ samples in a buffer and then trigger a transmission. As already introduced in Chapter 4, Broadcom Wi-Fi chips support two ways to implement raw signal transmissions. The first one works on all devices and stores samples in the small sample-play buffer that is normally filled with samples of tones used for calibration purposes. As the buffer can only hold up to 512 samples, we focus on the second approach in this chapter. It allows to transmit up to 131 072 samples on a suitable Wi-Fi chip such as the BCM4358. In this approach, we store the IQ samples in the Template RAM of the D11 core and trigger the transmission by setting the corresponding D11 registers.

For implementing the SDR transmissions, we faced the following challenges: (1) We had to understand which registers need to be written to setup and start a transmission, and (2) we had to choose the correct number format to store the IQ samples.

Below, we first describe our implementation in Section 9.1. Then, we document our experiments with our SDR implementation and analyze the transmitted signals using a WARP SDR in Section 9.2. We continue with a discussion in Section 9.3. As the SDR implementation has a lot of development potential, we propose future work in Section 9.4 and present related work in Section 9.5. Then, we conclude in Section 9.6.

*Wi-Fi chips have many undocumented capabilities that exceed the requirements for implementing a standard conform Wi-Fi transceiver.*

*Using Wi-Fi chips as software-defined radios allows to run experiments on hardware with realistic properties of commercial devices.*

*Broadcom chips can transmit up to 512 samples from the sample-play buffer and up to 131 072 samples from Template RAM.*

*We had code that only hinted us at the SDR transmission capabilities.*

*We present the implementation and an evaluation of the Nexmon SDR.*

## 9.1 IMPLEMENTATION

*Transmitting raw signals from Template RAM should work analogous to collecting samples.*

*D11 registers define where raw samples start and stop in Template RAM and another register controls the playback.*

During our analysis of various firmwares and drivers, we discovered a sample collect functionality in 802.11ac Wi-Fi chips created by Broadcom. It allows us to capture raw IQ samples of the digital-to-analog converters (DACs) into the Template RAM. We describe this functionality in Section 4.3.1 and illustrate the used system components in Figure 3 on page 21. In the initialization functions of this feature, we observed that the value of the `macbasedDACPlay` register controls whether the whole Template RAM or only half of it is used for sample collection. Hence, we concluded that these Wi-Fi chips might also be able to transmit samples from Template RAM. A closer inspection of the header files defining the registers of the D11 core revealed that there are `SamplePlayStartPtr` and `SamplePlayStopPtr` registers that likely define the sample-play address boundaries in the Template RAM. To finally trigger a transmission, we considered an operation analogous to triggering the collection of samples. To this end, one bit needs to be set in the `psm_phy_hdr_param` register for continuous sample collection and another bit can be set to stop after filling the assigned region in the Template RAM once. By trying other bit combinations, we observed that triggering a transmission from Template RAM works.

### 9.1.1 Raw sample transmission methodology

*As raw signals are too large to be part of a firmware patch, we load them during runtime using an `ioctl`.*

After experimentally verifying that raw transmissions work in general, we developed the following methodology to ease the use of raw transmissions. All steps shown below can either be called directly from a firmware patch or by sending `ioctl` messages to the firmware. We provide more information on communication with the firmware in Section 6.2.13. To prepare the transmission, we copy raw IQ samples into an unused region of the Template RAM. Each sample consists of two signed 16-bit values for the inphase (I) and quadrature (Q) components. We can either generate the raw samples in the firmware or create them in MATLAB and use an `ioctl` to copy them into Template RAM during runtime. Especially for long signals, it is not suitable to patch their samples into the firmware binary as they take up too much space in the patch memory region.

*After loading raw samples into Template RAM, we need to trigger their transmission.*

In the second step, we set transmission gains to manual control and store the boundary addresses of our signal in the `SamplePlayStart` and `StopPtr` registers. Then, we are ready to trigger the transmission by activating `macbasedDACPlay`, initiating an RX-to-TX sequence and starting to play back samples by writing into the `psm_phy_hdr_param` register. This step can either be performed in the ARM core, or as a reaction to a time critical event in the D11 core (e.g., as a quick answer to receiving a frame). After finishing a transmission, the Wi-

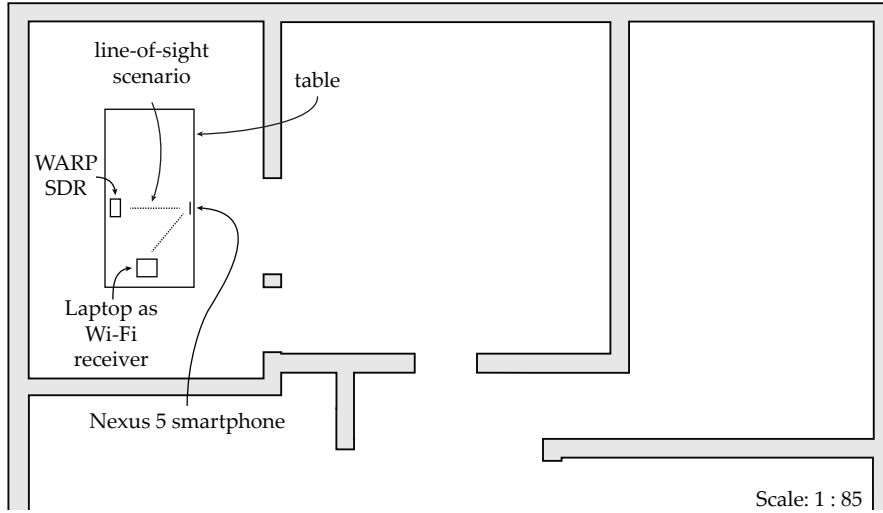


Figure 7: Experimental setup in an apartment in a rural environment with no other Wi-Fi traffic on channel 122 and 120. (based on [75])

Fi chip continues to transmit a carrier wave and the D11 core is not able to receive any new frames. To solve this problem, we have to stop the transmitter and reset the clear channel assessment (CCA) in the baseband. Implementing this in the D11 core allows us to integrate our transmitter into regular Wi-Fi communications and react to incoming frames. We published the source code as described in Section A.5 so that other researchers may build applications based on our implementation.

*To stop the transmitter and activate the receiver after a transmission finishes, we have to perform a CCA reset.*

## 9.2 EXPERIMENTS

In our experimental evaluation, we study the performance of our raw-signal transmitter and discuss the experimental results. As environment, we choose an apartment in a rural area with only low amount of Wi-Fi traffic so that our experiments are not affected by high levels of interference. As we do not focus on multi-path propagation, we simply stayed in one room. We illustrate our communication setups in Figure 7.

*We perform our experiments in a rural area to reduce the risk of Wi-Fi interferers.*

### 9.2.1 Experimental setup

To evaluate the raw signal transmission capabilities, we measure how well Wi-Fi frames transmitted from raw samples stored in the Template RAM can be received by an off-the-shelf Wi-Fi node in comparison to receiving the same frames transmitted through the regular Wi-Fi modulation chain. To this end, we first generate acknowledgment frames at all 802.11a/g rates in MATLAB and store the signals in a format that we can directly load into the Template RAM by ex-

*We first send a Wi-Fi frame stored in raw samples and then inject a Wi-Fi frame with the same modulation coding scheme.*

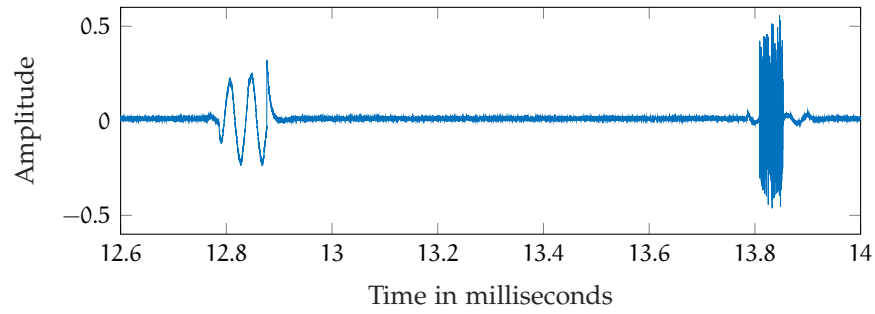


Figure 8: Generally, raw transmissions play back the IQ samples from the Template RAM resulting in the signal on the right. Especially, on crowded channels, we observed that sometimes only a carrier is transmitted but no other signal (left). (based on [75])

*The interval for sending raw samples is 3 ms, followed by a regular frame transmission after 1 ms.*

ecuting a script in the smartphone's user space. Due to the size of the raw signals, we cannot fit them directly into the firmware patch and, hence, run our script after loading our patched Wi-Fi firmware. In the firmware, we trigger the transmission of the raw signals every 3 ms and after a short break of 1 ms, we directly inject an acknowledgement frame through the regular frame transmission path. This frame is encoded with the same modulation settings and contains a similar content as our raw Wi-Fi frames.

### 9.2.2 Experimental evaluation

*Wi-Fi frames transmitted from raw samples can be received with a similar performance as regular Wi-Fi frames.*

Our experimental results show that frames transmitted through raw signal transmission have a similar reception performance as regular frames. In case of a crowded channel, it can happen that—instead of the raw signal—only a carrier wave is being transmitted. This effect happened very randomly in our experiments but seems to be avoidable on rather unoccupied channels. We illustrate this effect in Figure 8. On the left we show a transmission, where only the carrier is observable and on the right we show a correctly transmitted acknowledgement frame. Based on further investigations performed by Jan Ruge, signal transmissions become more stable by first setting the `psm_phy_hdr_param` to zero before setting the bits that trigger a transmission. This seems to avoid failed transmissions that only leak the carrier.

*Carrier transmissions before and after a raw signal transmission are avoidable by carefully timing the transmitter's activation.*

Additionally, in Figure 8 we can observe that a small carrier signal is visible before and after the transmitted acknowledgement frames. This is due to the fact, that we first perform the RX-to-TX sequencing and then start the sample playback from the Template RAM. By first triggering the playback of a signal that starts with a suitable number of zeros and then triggering the RX-to-TX sequencing, we can start playing back samples as soon as the transmitter is ready. To get rid of the small carrier after the signal transmission, we simply need to



initiate the CCA reset at the end of the sample transmission. This directly turns off the carrier.

### 9.3 DISCUSSION

In this chapter, we demonstrated that off-the-shelf Wi-Fi chips have much more capabilities than officially advertised by vendors. Using Wi-Fi chips in a way only software-defined radios could be used before, we open new scenarios for researchers that can transform ubiquitously available mobile devices such as smartphones into general purpose radio transmitters in the 2.4 and 5 GHz bands. With the ability to store raw IQ samples in a relatively large buffer it is now possible to add physical layers from other standards or even test new ones: this enhances cross-technology communication to an unprecedented degree of flexibility. The great availability of smartphones also enables the creation of large software-defined radio testbeds built on devices that are cheap, mobile, and always around the end users.

*Nexmon SDR turns ubiquitously available smartphones into fully-fledged software-defined radios for the 2.4 and 5 GHz bands.*

### 9.4 FUTURE WORK

Besides the transmitter, we should next focus on the reception of wireless signals, which is also possible on Broadcom chips using the sample collect feature. However, receptions are more complex as we have to consider setting the correct gains or configuring the undocumented automatic gain control hardware in the physical layer to receive signals with a usable dynamic range for the analog-to-digital converters (ADC).

*The receive path still requires investigation including automatic gain control features.*

#### 9.4.1 Controlling Wi-Fi chips from MATLAB

Another extension focuses on the use of smartphones as a replacement for WARP SDRs that are operated by MATLAB running WARP-LAB. In this setup, signals are generated in MATLAB and transferred into a buffer in the software-defined radio. Then the transmission is triggered on all sending nodes and the reception on the receiving nodes is triggered at the same time. To exchange frames between MATLAB and smartphones, we can connect smartphones over USB with a computer. Then, we activate USB tethering to exchange Ethernet frames over USB with the phone. This setup allows to establish a UDP or TCP based communication link between the connected smartphone and MATLAB so that frames and control commands can be exchanged. On the phone, a UDP-to-IOCTL proxy would receive ioctl messages sent by MATLAB over UDP and send them to the Wi-Fi chip.

*MATLAB can communicate to a smartphone using UDP datagrams tunneled over USB to send ioctls to control the firmware.*

### 9.4.2 Comparing Nexmon SDRs to WARP SDRs

*Smartphones are cheap SDRs with 80 MHz bandwidth.*

*Broadcom chips only support single-antenna applications without FPGA support.*

*We created a ping-pong buffer based WARPLAB extension to continuously refill and read buffers in the FPGA while transmissions and receptions are ongoing.*

A smartphone-based SDR testbed would be much cheaper than using WARP SDRs and it would support 80 MHz bandwidth. However, it lacks some of WARP's advanced features. First, using Broadcom Wi-Fi chips, only one antenna can be used for raw signal operation. Hence, no MIMO applications are supported. Second, Broadcom Wi-Fi chips have no FPGAs included to optimize wireless implementations with hardware support. Third, the WARPLAB implementations before release 7.5.0 were only able to hold up to 32 768 raw samples, which is less than the 49 152 samples we can store on BCM4339 chips. However, we created a proof-of-concept implementation for WARP SDRs that uses the 2 GiB DRAM on the WARPs to store samples. With this extension, WARP SDRs can collect and transmit multiple seconds of raw signals which exceeds the capabilities of Broadcom's Wi-Fi chips. Our implementation was the foundation for the 7.5.0 release of WARPLAB<sup>1</sup> that also supports using the DRAM.

Our WARPLAB extension uses a ping-pong buffer approach that splits the transmission buffer in two halves. We first fill the buffer completely and start the transmission from the beginning. Whenever the end of the buffer is reached, we loop back to its beginning and continue transmitting. During the transmission, we constantly check, whether the transmission of one half of the buffer finished. Whenever this is the case, we refill this half of the buffer with new samples while the transmitter operates on the other half. As long as we can refill the buffer faster than samples are transmitted, we can continuously transmit samples. Continuous receptions work accordingly. On the WARP SDRs we use this principle to fill the small buffers in the FPGA with samples stored in DRAM. For a complete continuous operation, we would also have to refill the DRAM with samples coming in over Ethernet.

### 9.4.3 Implementing continuous transmissions and receptions

*First experiments show that DMA controllers could be used to refill the Template RAM continuously during a raw signal transmission.*

Under the assumption that the Template RAM can be read and written at the same time, we could implement a similar ping-pong buffer based system on Wi-Fi chips to transmit and receive signals continuously. For transmissions, it would be most efficient to transfer samples either directly from the host's RAM (on PCIE connected chips) or the ARM's RAM (on SDIO connected chips) into the Template RAM by using a DMA controller. Somehow, in our experiments with the DMA controller, only the first transmitted frames ended up in Template RAM. As soon as this region is full, no more frames are stored there until the D11 core restarts. By understanding how to transfer

<sup>1</sup> WARPLAB 7.5.0 release notes: <https://warpproject.org/trac/wiki/WARPLab/Downloads?version=68>

arbitrary amounts of bytes into Template RAM, we could implement a ping-pong buffer that can be refilled quickly enough for continuous transmissions. For receptions, we did not find a way to initiate DMA transfers from Template RAM, yet. Nevertheless, the D11 core can trigger a quick copy of data from Template RAM into the shared memory, from where we can perform DMA transfers similar to the ones we used to extract channel state information (CSI) in Section 8.1.2. Using this approach, we could use the large internal memory of a smartphone to store raw samples that exceed the amount storable in the WARP SDRs DRAM. If we further managed to exchange data sufficiently quick over USB, we could use smartphones as SDRs with continuous transmission and reception capabilities that could be connected to real-time capable SDR software such as GnuRadio.

*To collect received samples, we could first copy them into the shared memory and then trigger DMA transfers to the ARM's RAM or directly to the host.*

#### 9.4.4 New applications on Wi-Fi chips

Besides the ideas to extend the Nexmon SDR, it can also be used as basis for new applications on smartphones. Transmitting and receiving arbitrary signals with a Wi-Fi chip revolutionizes cross-technology communication (CTC) schemes. We no longer need to generate waveforms for other communication standards by choosing Wi-Fi payloads that result in a suitable waveform, for example, to transmit ZigBee frames as done by Li and He in [57]. Instead, we can simply create clean signals as IQ samples and transmit them in the Wi-Fi bands or analyze incoming signals of other systems. Crawling driver source files also revealed that Broadcom Wi-Fi chips could be used as spectrum analyzers or for ultra low bandwidth (ULB) communications. Those use either 2.5, 5 or 10 MHz bandwidth which could significantly reduce the throughput required to continuously exchange IQ samples with the Wi-Fi chip. In Chapter 13, we present another new application for smartphones. We use the raw signal transmission capabilities to generate Wi-Fi frames with embedded secret information to create a covert channel between two smartphones.

*Cross-technology communication and covert channels are only two out of many applications realizable using the Nexmon SDR.*

*Spectrum analysis and ultra low bandwidth capabilities still need further investigation.*

## 9.5 RELATED WORK

To the best of our knowledge no other work exists that converts Wi-Fi chips into software-defined radios (SDRs). The RTL-SDR project [67], however, managed to convert DVB-T dongles into SDR receivers with a low bandwidth of up to 3.2 MHz but a very wide frequency range that goes from 22 MHz up to 2.2 GHz depending on the device. In the DVB-T dongle, the SDR mode was intentionally implemented to receive FM radio around 100 MHz but could be extended to the full frequency range supported by the hardware.

*RTL-SDR turns commercial DVB-T dongles into SDR receivers.*

The first project that converted a Raspberry Pi into a radio transmitter is PiFm [63]. It was initiated by Oliver Mattos and Oskar Weigl

*By abusing PLLs connected to GPIO pins in the Raspberry Pi, one can not only generate frequency modulated or single-side-band signals, but also transmit IQ samples at a carrier frequency.*

*Wide-spread SDR platforms in the research community are, among others, USRPs, WARPs and HackRFs.*

*GnuRadio is the basis for software implementations of various communication systems.*

*Open access to the implementation allows easy extensions and security analyses.*

and allows to transmit frequency modulated signals through a wire antenna connected to a GPIO pin. The pin is connected to a phase-locked loop (PLL) and can be used to output a pulse-width modulated (PWM) waveform that can mimic an FM signal transmitted between 1 and 250 MHz. The project was extended [40] by Richard Hirst who introduced the use of a DMA controller to write signals quickly and well timed into hardware registers. Dolle G. Ten introduced another extension. Instead of transmitting FM signals, he managed to create single-side-band (SSB) signals using the Raspberry Pi's PLL. To gain even more flexibility, Evariste Courjaud created a quadrature modulator to transmit IQ samples and thereby support a variety of modulations. His project RPiTX [22] can be operated between 5 kHz and 500 MHz and thereby cover a lot of applications.

Besides the projects mentioned above, many commercial SDR platforms exist, that beat the cheap platforms in bandwidth, frequency range and MIMO capabilities. Additionally, dedicated SDRs often contain an FPGA on which we can implement time-critical signal processing steps in both the transmitter and the receiver. The most well known platforms used by the research community are the universal software radio peripherals (USRPs) and the wireless open access research platforms (WARPs) that can be acquired at moderate prices. All of them include an FPGA and come directly with MIMO support or can be connected to synchronously operate with multiple antennas. In the low price sector we can find the HackRF One which is limited to 20 MHz bandwidth and has no FPGA.

Based on software-defined radios, many proprietary communication systems were reimplemented in an open fashion. GnuRadio is a wide-spread toolkit that collects many signal processing building blocks that can be used to implement baseband processors in software. Among the many projects, one can find implementations to receive and decode GSM signals (GR-GSM, Airprobe), LTE signals (GR-LTE), Wi-Fi signals (GR-IEEE802-11), and automatic dependent surveillance – broadcast (ADS-B) signals (GR-ADSB). One can also find projects for transmitters that support basic analog modulations such as FM, AM, or SSB or even more complex system such as the digital television standard DVB-T2 (GR-DVB). Based on those open implementations, researchers can better understand how those standards work. They can develop improvements for the next generation or find security vulnerabilities in the upper layers more easily. To this end, it helps to be able to inject arbitrary bits with the correct physical layer implementation.

## 9.6 CONCLUSION

In this chapter, we presented how we turned Wi-Fi chips installed in commercial off-the-shelf smartphones into arbitrary waveform trans-

mitters. This allows them to be operated in a similar way as software-defined radios and directly interwork with other communication technologies. In Chapter 13, we use our Nexmon SDR in a controlled environment to transmit modified Wi-Fi frames into which we embedded covert information by applying custom filters to the baseband signals. This chapter also concludes the part on tools and we continue to present applications in the next part.

*The Nexmon SDR turns Wi-Fi chips into software-defined radios that can transmit arbitrary waveforms.*

## 9.7 MY CONTRIBUTION AND ACKNOWLEDGEMENTS

After detecting Broadcom's sample collect feature and an initially not working MAC-based sample-play option during joint experiments with Francesco Gringoli, I expected the existence of a way to transmit IQ samples from Template RAM. Based on this expectation, I performed experiments until I discovered how the MAC-based sample-play feature worked. Subsequently, I implemented the Nexmon SDR tool.

*Perseverance made it possible to get the Nexmon SDR working.*



## Part IV

### APPLICATIONS

We create applications based on the Nexmon framework presented in Part III. We start with a simple ping offloading application to show the reduced energy consumption and the reduced latencies of applications running on a Wi-Fi chip in Chapter 10. Then, we propose how applications can set requirements for physical layers to build wireless software-defined networks in Chapter 11. Switching to applications in the security domain, we present a reactive jammer for Wi-Fi systems that illustrates real-time and basic software-defined radio capabilities in Chapter 12. Another advanced application is our covert channel presented in Chapter 13. It hides information by prefiltering outgoing Wi-Fi frames. It uses the Nexmon SDR (see Chapter 9) to transmit frames and the Nexmon CSI Extractor (see Chapter 8) to extract covert information. Last but not least, we present applications using the Nexmon framework on other platforms in Chapter 14.





To demonstrate the benefits of modifying firmwares with Nexmon, we chose a simple ping-offloading application whose source code we published as described in Section A.2. Instead of answering ping requests in the kernel, we do it in the firmware. We chose the ping application as it receives a frame, analyzes its contents and then “forwards” it back to the transmitter. Instead of sending it back to the transmitter, one could also forward it to another node in the network. Hence, the performed operation is similar to a mesh node that receives and forwards frames. As efficient mesh specific implementations require changes to the kernel, we use the kernels ping implementation to mimic the operation of receiving and forwarding frames. This allows us to keep the experimental setup simple, but still compare our firmware implementation to an efficient frame forwarder available in every kernel.

Below, we first describe our implementation in Section 10.1 and then continue with an experimental evaluation in Section 10.2, where we focus on our implementations general operation, its energy consumption and its delay. At the end, we present a discussion in Section 10.3, related work in Section 10.4 and a conclusion in Section 10.5.

### 10.1 IMPLEMENTATION

To simply answer pings by the operating system’s kernel, we do not need to perform any changes. To offload the operation to answer ping requests into the Wi-Fi firmware, we have to insert a hook in a function that handles frames before they are sent to the host. There, we can analyze the frames and answer incoming ping requests. To this end, we hook the function that handles offloading the address resolution protocol (ARP) in the `wl_sendup` function that is called after replacing Wi-Fi headers by Ethernet headers, shortly before pushing up frames to the host. Here, we check for ping requests and generate ping responses encapsulated in Ethernet frames that we send using the `wlc_sendpkt` function that creates the correct Wi-Fi headers and transmits the frames. By using functions that handle Ethernet frames in the Wi-Fi firmware, we can simply concentrate on our protocol implementation and ignore details of the MAC and physical layers. This includes the current state of the Wi-Fi connection as well as the topology, so that our application works no matter we encrypt our traffic or not and no matter we use an ad hoc or a mesh topology to directly

*Answering pings in the firmware should be faster and more energy efficient than in the kernel.*

*The ping application mimics a frame forwarder.*

*We evaluate power consumption and processing delay.*

*We let the firmware convert between Wi-Fi and Ethernet frames first before handling the ping frames to abstract from the internal state of the Wi-Fi connection.*

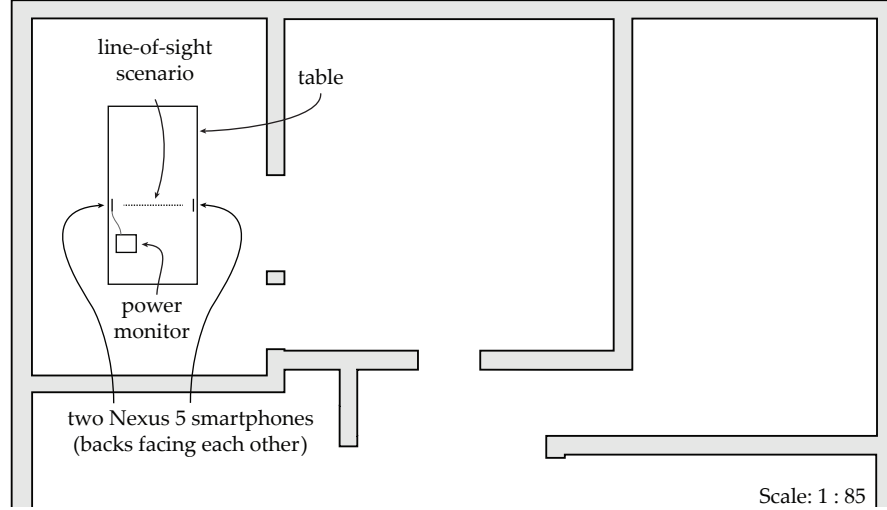


Figure 9: Experimental setup in an apartment in a rural environment with no other Wi-Fi traffic on channel 6. Both smartphones are installed on car mount holders to enhance the antenna radiation characteristics. (based on [75])

communicate with pairs of nodes. In the next section, we evaluate this implementation.

## 10.2 EXPERIMENTAL EVALUATION

We setup our experiments using two Nexus 5 smartphones. They both run the rooted Android stock firmware version M4B3oZ and are located on a table one meter apart as illustrated in Figure 9. We connected them using ad hoc mode on the otherwise unused channel 6 with 20 MHz bandwidth. They exchange 802.11ac frames with MCS 8. This is normally not supported in the 2.4 GHz band, but still available due to Nexmon. To send exactly only one frame per ping request and reply, we disabled retransmissions and AMPDUs. The latter would, otherwise, aggregate multiple ping packets into one Wi-Fi frame. This would falsify our results as we intend to compare the performance of handling single frames.

Using the Android debug bridge (ADB), we setup the first phone to transmit ping requests with 1200 byte payload to the second phone. For our experiments, we used ping intervals between 6 and 1000 pings per second. During our experiments, we can toggle this ping-offloading functionality by using an ioctl. To measure the power consumption with disabled or enabled ping-offloading, we attached a Monsoon Power Monitor to the battery ports of the second phone. During our initial measurements, we observed high power consumption peaks every 640 ms that even show up with disabled Wi-Fi chip and that disturb our measurements. Further experiments revealed that those peaks are generated by the LTE chip. By deactivating the

*We connect two  
Nexus 5  
smartphones in ad  
hoc mode and start  
sending ping  
requests.*

*We initiate between  
6 and 1000 ping  
requests per second.*

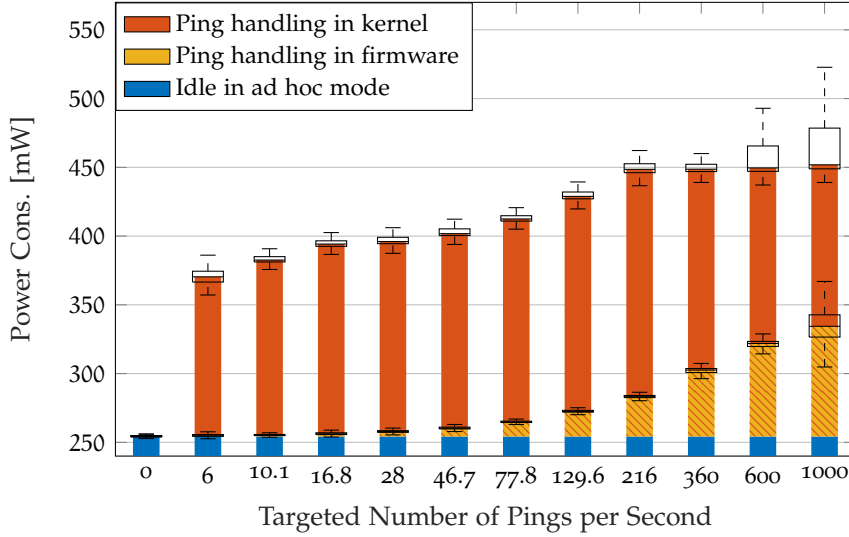


Figure 10: Operating ad hoc mode consumes 254 mW. Handling pings in the firmware smoothly increases power consumption, while handling frames in the kernel leads to a sudden increase with high variations. (based on [77])

LTE driver in the kernel, we get rid of the peaks and, hence, cleaner power consumption traces. To rebuild the kernel we used, see Section A.8.

For our evaluation, we performed three measurements always comparing the performance of our firmware ping-offloading implementation to the standard kernel implementation. First, we analyze the power consumption in Section 10.2.1, then we measure the number of actually transmitted ping requests in Section 10.2.2 and finally, we compare the round-trip times (RTTs) in Section 10.2.3.

### 10.2.1 Power consumption

In Figure 10, we present our power consumption results. The vertical axis displays the number of pings we initiated at the transmitter. This number can differ from the number of actually transmitted frames as well as the number of actually processed frames in the receiver, as frames can be dropped at various places on the path when storage and processing queues overflow. The horizontal axis shows the power consumption in milliwatts. It starts slightly below the 254 mW that the smartphone consumes by operating the Wi-Fi chip in ad hoc mode. As long as the smartphone's main processor is inactive, only between 10 to 20 mW are consumed by components that are not Wi-Fi related. The rest is mainly required to constantly listen for incoming Wi-Fi frames. The amount of energy required to handle only a few ping requests per second in firmware is negligible but increases when more ping responses have to be transmitted. Handling frames

*By deactivating LTE, we get clearer power consumption traces.*

*We evaluate three characteristics of our ping-offloading implementation.*

*Waking up the host's processor for handling pings requires at least 116 mW more energy than handling pings in firmware.*

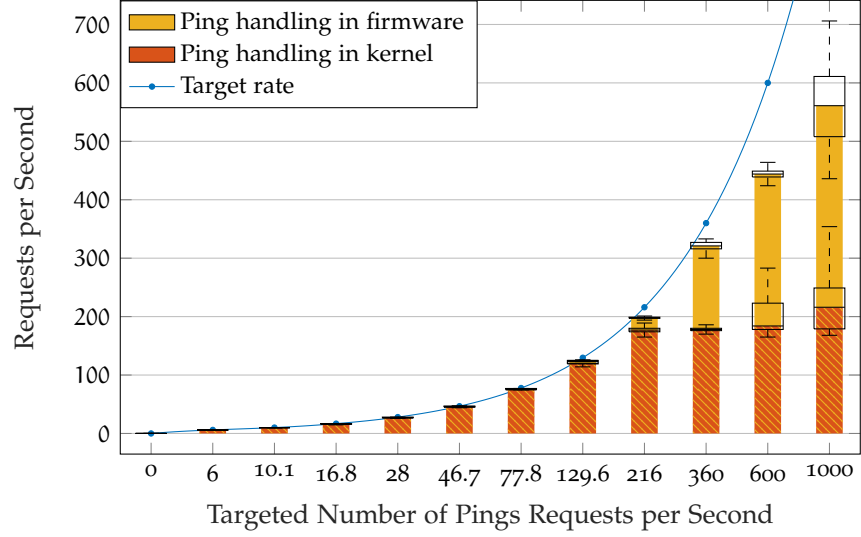


Figure 11: The numbers of actually transmitted ping requests stay below their target numbers, especially when handling pings in the kernel instead of the firmware. (based on [77])

*We need to further investigate, why the power consumption saturates for high ping rates.*

in the kernel, on the other side, always requires to wake up the main processor from its idle state. Hence, even for handling low numbers of frames, it abruptly increases by 116 mW. This increase reaches up to 194 mW for higher frame rates. Unexpectedly, the power consumption saturates starting from roughly 216 requests per second. In the following section, we investigate why this happens.

#### 10.2.2 Number of actually transmitted ping requests

*Power consumption stagnates as less pings are exchanged in the kernel compared to the firmware implementation.*

To analyze why the power consumption of handling frames in the kernel stagnates for high frame rates, we analyzed the number of actually transmitted ping request frames as this number may differ from the number we define in the ping program initiating the requests. We realized that the number of transmitted frames stays below the targeted number for both the firmware and the kernel operation when more than 200 pings per second should be transmitted. We illustrated our results in Figure 11. Due to the fact that every ping request is transmitted in a separate Wi-Fi frames and the ping application waits for a ping reply before transmitting the next ping request, the number of transmitted frames reduces when frames are dropped at the receiver who can only handle a limited number of frames per second. As processing frames consumes less energy than transmitting frames, the power consumption stagnates when less frames are transmitted. While the kernel implementation can only handle around 220 pings per second, the firmware processes 2.5 times more (around 560). In the next section, we analyze how quickly we can expect an answer by either the kernel or the firmware.

*The firmware can handle 2.5 times more ping frames per second than the kernel.*

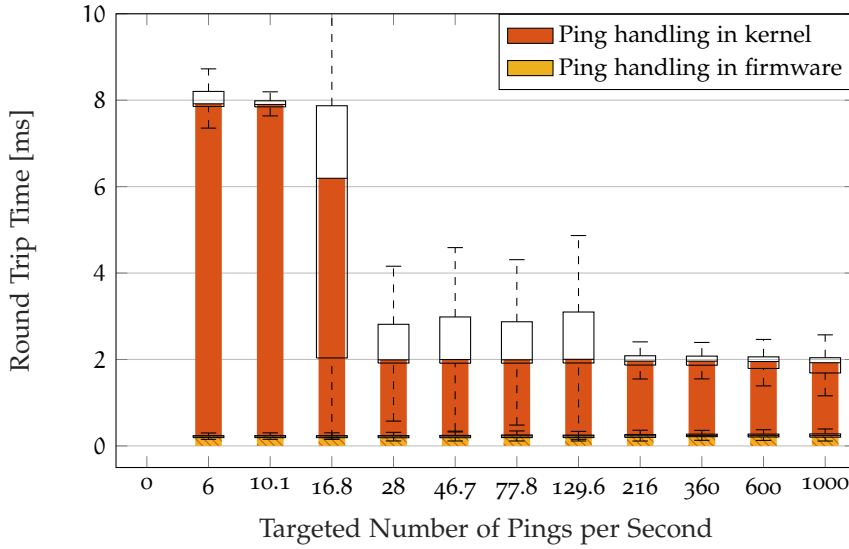


Figure 12: The round trip time to answer pings in the firmware is deterministically low at  $230\ \mu\text{s}$ , while it strongly varies and is much higher in the kernel, likely due to waking up from energy-saving states. (based on [77])

### 10.2.3 Round-trip times (RTTs)

Our firmware patch also outperforms the kernel implementation with respect to round trip times (RTTs), as illustrated in Figure 12. To get more exact RTT measurement results, we ignored the path through the Wi-Fi chip, kernel and user space to the ping application as scheduler implementations in the operating systems can have a high influence on when the ping application finally processes the received frames. Instead, we measured RTTs by subtracting the timestamps of ping requests and replies captured by a third node, in our case a laptop running in monitormode. Our results show that the firmware deterministically achieves  $230\ \mu\text{s}$  RTTs, while the kernel has RTTs between 6 and 8 ms for handling less than 28 frames per second and RTTs around 2 ms for higher frame rates. The high RTTs likely result from the fact that the kernel may fall into energy-saving mode between processing pings at low frame rates. Hence, it needs to wake up for every single ping request which takes extra time.

*Sleep cycles between ping requests at low rates hinder the kernel implementation to react quickly.*

*While the firmware sends responses in less than  $230\ \mu\text{s}$ , the kernel needs at least 2 ms.*

## 10.3 DISCUSSION

With our simple ping-offloading experiment, we demonstrated that firmware implementations are not only more energy efficient than kernel implementations. Their response times are also lower and deterministic, which results in higher frame handling rates. Those are important for low-latency applications, for example, to control machines. Having round trip times of  $230\ \mu\text{s}$  allows to answer al-

*The firmware implementation has deterministic and low latencies as well as a low energy consumption.*

most nine times faster than the kernel implementation at high frame rates. Further improvements could be achieved by answering frames directly from the D11 core, similar to sending acknowledgments from the template RAM. Programming the D11 core in Assembler is, however, less comfortable than writing programs for the embedded ARM processor.

#### 10.4 RELATED WORK

Most related to our application are applications that enhance network performance by offloading simple but computation expensive operations into specialized hardware or processors, such as TCP checksum computations. The survey in [15] created by Chase, Gallatin, and Yocum states that TCP checksum offloading together with copy avoidance yields roughly 70 percent in throughput improvements. In our application, we were even able to increase the number of actually transmitted and answered ping requests by 150 percent (factor 2.5), as presented in Section 10.2.2. Also Broadcom Wi-Fi firmwares come with installed TCP offloading engines (TOEs).

Besides TOEs, Broadcom firmwares also offer ARP offloading. In [69], Peng et al. tackle the problem of broadcast frames in Wi-Fi systems that wake up a phones operating system and increase power consumption even though not all broadcast frames are important for a receiver. They also show that many smartphones already employ an ARP offloading engine in the Wi-Fi firmware that allows to handle ARP requests directly in the firmware without waking up the main processor. This also matches our results in Section 10.2.1 that handling frames in the firmware is more energy efficient than handling them in the main processor.

Even more optimized are frame processing solutions in FPGAs. In [66], Mühlbach and Koch present the NetStage platform that employs reconfigurable computing for high-throughput low-latency network processing. Based on this platform, they implemented a honeypot whose performance exceeds that of many software solutions. Additionally, it is resilient against code injection attacks due to the lack of software-executing processors, which is very beneficial for such security critical applications. In [25], Engel and Koch focus on implementing a reconfigurable computing unit into wireless sensor network (WSN) nodes to support computationally-intensive distributed applications on low-power devices.

Especially in industrial control applications, quality of service and in particular low round-trip times are required to move control nodes away from machines and place them into a network. Saifullah et al. model the corresponding end-to-end delays in [73] for wireless networks. With measured round-trip times of 230  $\mu$ s, our firmware

*TCP offloading engines are often used to release the main processor from spending time on calculating checksums.*

*ARP offloading engines avoid waking up the main processor from energy-efficient idle operation mode for handling ARP requests.*

*Using FPGAs, processing tasks can be offloaded to further enhance security and reduce power consumption.*

*Deterministic, low round-trip times are required by industrial control applications.*

implementation is able to meet sub-millisecond delay requirements of control applications even in wireless multi-hop networks.

## 10.5 CONCLUSION

In this chapter, we presented a ping-offloading implementation for Broadcom Wi-Fi chips. Even though, it does not meet the requirements of a regular ping application that measures the whole network path between to end stations including the delays in the communication stacks, it allows to mimic and compare frame forwarding applications without custom kernel extensions. We have shown that the firmware implementation's power consumption only slightly increases with an increasing number of ping requests, while the main processor abruptly consumes much more power as soon as the first ping packets require handling. Additionally, processing frames on the main processor takes much longer than handling them in the firmware. This leads to both lower frame processing rates and higher round-trip times. In the next chapter, we present how more advanced video streaming applications can be enhanced with a software-defined networking approach implemented in the firmware.

*Our ping-offloading implementation is energy efficient and has high frame processing rates at low round-trip times.*

## 10.6 MY CONTRIBUTION

The need for a simple application to evaluate the benefits of modifying Wi-Fi firmwares made me come up with the ping-offloading application. I also performed the implementation and the evaluation.





## SOFTWARE-DEFINED WIRELESS NETWORKS WITH FLOW-BASED PHY CONTROL

---

We claim that various applications that communicate directly between nodes could be improved by giving an application more control over physical-layer parameters. The easiest would be the modification of the modulation coding scheme used for broadcast transmissions. Depending on the intended receivers, it makes sense to either increase the throughput but only reach close receivers, or enhance robustness of the transmitted frames and reach larger areas. By, additionally, adjusting transmission power and beamsteering, we could focus our radio energy at our receivers and reduce interference with other communication parties. This is especially important in dense environments, where many nodes communicate simultaneously on the sparse frequency spectrum.

Going one step further into reconfigurable radio transceivers, one can not only modify parameters of one communication standard. Instead, one can freely switch between various standards depending on which one best fits the application requirements at hand. For example, Bluetooth low-energy (BLE) or ZigBee can operate their transceivers at very low power but they also offer only low throughput. Wi-Fi, on the other side, offers the highest throughput we can currently achieve in commercial wireless devices, but it consumes significantly more energy, especially in ad hoc communication topologies, where the receiver is always active. As discussed in Section 10.2.1, operating a Nexus 5 smartphone in ad hoc mode continuously consumes at least 254 mW. For mobile applications with limited battery capacities this is quite high, especially, as it only includes the power for keeping the receiver active. Transmitting information requires additional energy. It would be more efficient to use BLE or ZigBee to manage the ad hoc connection and only turn Wi-Fi on, when data should be exchanged. It would be most efficient to handle all these communication standards in a single-chip solution. This allows to pass information directly between the different physical-layer implementations without pulling the device's main processor out of an energy-saving idle mode. Doing the latter would be unfavorable as presented in Section 10.2.1. Broadcom Wi-Fi chips at least already offer single-chip solutions for Wi-Fi and Bluetooth for mobile systems.

Another growing application is video streaming. According to [16], video data is expected to grow 14-fold between 2013 and 2018—outpacing the general data traffic growth—and amounting to close to 70% of all mobile data traffic. This growth cannot be handled by

*Control over physical-layer parameters should be beneficial to various applications.*

*Both parameter modifications and the ability to switch between standards together lead to optimized energy consumption and performance.*

*Low-power communication technologies could be used to control ad hoc connections, while Wi-Fi is turned on only for transmitting large chunks of data.*

*Video streaming traffic is constantly growing.*

*Physical-layer control can help to optimize the transmission of videos that were encoded into multiple quality layers.*

advances in wireless network technology such as long-term evolution (LTE) or LTE-Advanced alone. Especially, for device-to-device video streaming broadcasts, it can be beneficial to optimize physical-layer parameters. In case one video stream should be broadcasted to many receivers, we could apply scalable video coding (SVC) to generate multiple video streams at different quality levels. To deliver the base quality to every receiver in range, one could use a robust transmission at high power. Due to the low throughput of this mode, not all layers can be distributed using this setting. Instead, we could focus on broadcasting higher quality layers only to stations that are able to receive less robust signals and use the lowest transmission power possible to achieve this goal.

*We can gain access to physical-layer parameters and even more flexibility by attaching software-defined radios to Android smartphones.*

To implement those applications, we need to gain access to lower communication layers. We realized this line of work before Nexmon by connecting software-defined radios to our smartphones and using them as communication interfaces. The FPGA included in the WARP SDR allows to efficiently implement various communication standards directly in hardware so that the device can act as an Ethernet-to-(almost)-Anything bridge. This is very similar to the Broadcom Wi-Fi chips installed in smartphones, as they also exchange Ethernet frames with the host's operating system and create valid Wi-Fi frames on the MAC and PHY layers internally.

*Using Nexmon, we gain access to various physical-layer parameters.*

The WARP-to-Android solution motivated our work on Nexmon, as the former solution is not suitable for realistic usage scenarios of smartphones that also require mobility. As Nexmon enables fine grained access to physical-layer parameters, we also consider it for handling simple per-frame or per-flow parameter adjustments and further enhance energy-consumption and performance by offloading the filtering of flows and choosing of the desired physical-layer parameters into the Wi-Fi firmware.

*Using flows instead of per-frame control reduces complexity and enhances portability.*

As example application for using those features, we implement a simple MJPEG-based scalable-video codec that supports streaming layered video on Android. As all frame transmissions of each video layer have the same physical-layer requirements, it is useful to bundle them in flows and then define requirements for those flows. The requirements do not directly define the physical-layer parameters. Instead, they only define the flows' properties and the flow filter separately decides which physical-layer settings best fit the needs to fulfill those requirements. This way, the application designer does not need to know the details of all physical layers his application can run with. The resulting system brings software-defined wireless networking (SDWN) to a widely available platform—namely smartphones.

*Fast scalable-video coding and access to PHY parameters was required.*

Developing our physical-layer supported scalable-video streaming system, we faced the following challenges: (1) We needed a working scalable-video codec for Android that gives easy access to the generated quality layers, (2) we had to get access to physical-layer

parameters when transmitting frames, and (3) our solution had to be real-time capable.

In the following sections, we first present the design of our flow-based control system that allows to change physical-layer parameters according to application requirements—in particular our scalable-video codec—in Section 11.1. We continue with the description of our practical implementation in Section 11.2, where we focus both on the existing WARP-based implementation as well as our current work-in-progress solution that uses Nexmon. In Section 11.3, we evaluate our WARP-based implementation and continue with a discussion in Section 11.4. In Section 11.5, we first present future work, followed by related work in Section 11.6. Then, we conclude in Section 11.7.

## 11.1 DESIGNING A SDWN SYSTEM FOR SMARTPHONES

In this section, we present the design of our software-defined wireless networking (SDWN) system using both the Android-to-WARP-based solution and the Nexmon-based extension using the internal Wi-Fi chip to gain control over physical-layer parameters. We also describe how to implement flow filters for both approaches as well as the design decisions that lead to our custom scalable-video codec.

### 11.1.1 System overview

Regarding hardware, our system is designed for smartphones that are either connected to WARP SDRs or that use a Nexmon-capable internal Wi-Fi chip. The smartphones represent mobile end-devices running applications. The Wireless Open-Access Research Platforms (WARPs) offer flexible physical-layer implementations that run in real-time on field-programmable gate arrays (FPGAs). In our system, we bring both devices together to allow applications the setup of physical transmission requirements on a flow basis. This level of flexibility is generally not provided by off-the-shelf wireless chips that implement standard-compliant communication mechanisms, leaving little room for research outside the bounds of the standards. However, Nexmon tackles those limitations by modifying the proprietary firmware that shields users from controlling detailed parameters for Wi-Fi transmissions. Hence, we use Nexmon in this chapter as an alternative to connecting SDRs to a smartphone.

A flexible physical-layer implementation can be controllable by applications that intend to optimize their data transmissions. However, full control is infeasible when multiple applications require access to the physical layer. Additionally, each application would need separate optimization strategies for different physical layers. We, hence, introduce an abstraction layer that takes control over setting physical parameters according to the applications' needs (see Figure 13).

*We present both an implementation using WARP SDRs as well as one based on Nexmon, but focus our evaluation on the SDR setup.*

*We implement our flow-based SDWN system by using either WARP SDRs or the internal Wi-Fi chip of a smartphone.*

*SDRs offer capabilities that are generally not accessible in off-the-shelf wireless chips.*

*Nexmon unleashes capabilities that we need for our implementation.*

*We introduce an abstraction layer that hides physical-layer details from the application behind selectable requirements.*

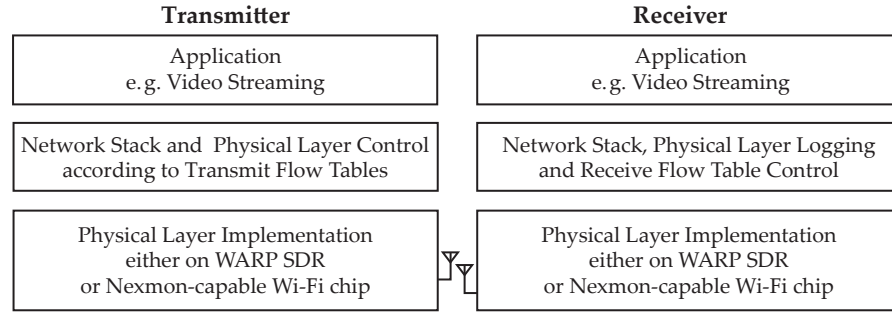


Figure 13: Abstract system overview. (based on [76])

*We differentiate flows by TCP and UDP ports.*

To handle different connections, we take advantage of the concept of flows. Flows can, for example, be differentiated according to TCP or UDP ports. Applications can then define abstract properties per flow such as robust packet delivery or high throughput. Our intermediary service translates these requirements to physical layer properties such as modulation scheme, transmit power or transmission priority. This way, we can handle multiple applications and abstract from lower-layer details. In the following, we describe our system in more detail.

#### 11.1.2 Overview of the system components

*The Nexmon-based solution is simpler to implement, as it does not require tunnelling frames to an SDR.*

Depending on the used physical-layer components, the proposed system consists of either two or four components. In case of connecting a WARP SDR to the smartphone, we need: (1) a scalable video codec (SVC)-based video streaming application as a practical example application, (2) the WARP “VPN” Service that allows to create tunneling devices (TUN) under Android to tap and inject IP packets without root privileges, (3) an Ethernet connection to the WARP to tunnel packets to the lower layers, and (4) the physical and data link layer implementations in the WARP. In case of using an internal Nexmon-capable Wi-Fi chip, we can get rid of the WARP “VPN” Service as no tunneling over Ethernet is required. We simply need: (1) the SVC-based video streaming application, and (2) a patched Wi-Fi firmware to gain access to all outgoing packets. In the following, we describe the components in detail.

#### 11.1.3 Enhancing SVC-video streaming

*We focus on scalable-video coding.*

Motivating example for our application-centric physical layer is the increasing demand for reliable, and network-efficient video streaming. Due to requirements for flexibility and opportunistic delivery coming from the heterogeneity of video formats, playback devices and network throughput variations, we focus on scalable-video coding. Compared to other data streams, video streams do not rely on

error free transmission and can cope with a certain number of bit errors and packet loss. The latter might lead to degradation of the image quality, but it still meets real-time requirements. SVC additionally offers to encode video streams into multiple quality layers. The base layer is always required to play the video. Higher layers that lead to better image quality, resolution or frame rate, are optional. If the physical layer can differentiate, which data streams are more important for the application than others, it can allocate the appropriate amount of resources to fulfill the application's requirements. For example, by using more robust modulation schemes and different transmit powers, the system can save energy at the transmitter, while maximizing quality at the receivers.

*Scalable-video codecs encode videos into different quality layers that we intend to transmit with different requirements on robustness or throughput.*

#### 11.1.4 Complete system overview

In Figures 14 and 15, we present our complete system. In Figure 14 we illustrate the transmitter that encodes a video stream and uses our system to transmit it to the receiver illustrated in Figure 15, which decodes and plays back the video stream. The encoder splits the video stream into three different quality layers. In Figure 15, we show how the videos would play back if only the base, the base and the first layer or the base and both extra layers were received. To transmit the data streams of each layer, the application could generate headers for all layers up to the application layer on its own and could either tunnel the resulting frames over Ethernet to the WARP (Option 1) or inject them using Nexmon-capable Wi-Fi chips (Option 2). Even though, this gives an application full control over the network stack and allows frame-wise settings for the physical transmissions, it also complicates the access control to the physical layer, especially if multiple services need access to it at the same time. Also, system-wide services such as name and address resolution would be application depended.

*Applications could generate and inject raw Wi-Fi frames to gain control over the lower layers.*

*This tight per-frame control complicates access to the physical layer especially when multiple applications run in parallel.*

#### 11.1.5 Isolating the application from physical-layer settings

To avoid these complications, we decided to use the existing Linux kernel to handle transport and network layers. This gives all IP-based applications access to our implementation and it allows to use existing transport layer protocol implementations such as UDP and TCP. To get access to IP packets on Android devices, we consider two options. For the Nexmon-based approach, we can simply extend the Wi-Fi firmware and inspect all outgoing frames to detect those we want to handle specifically. For the SDR-based approach, we instantiate a virtual private network (VPN) service to get a file descriptor on a TUN device. This VPN setup has the side effect that we can define IP subnetworks that are handled by our application-centric physical

*It is beneficial to use the existing network stack implementations for handling UDP or TCP over IP.*

*To gain access to the generated IP packets, we either need a VPN service or a modified Wi-Fi firmware.*

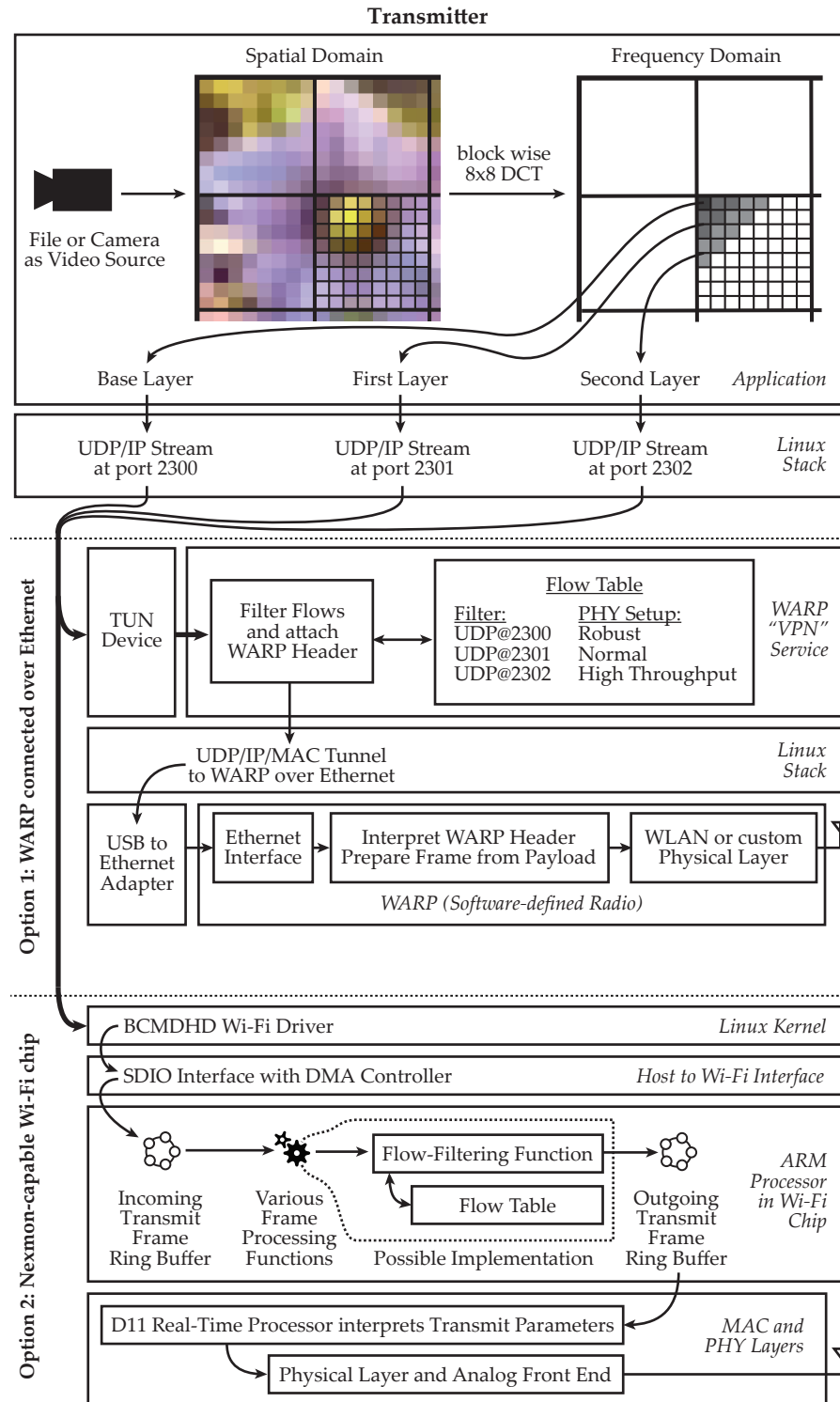


Figure 14: System overview that shows the components of our transmitter. At the top, we illustrate our example application—a scalable video codec. It uses regular user datagram protocol (UDP) over Internet protocol (IP) communication offered by the Linux kernel. Below, we either use the WARP “VPN” Service to setup physical layer requirements according to application-controlled flow tables and send frames using WARP SDRs or implement flow tables into the firmware of Nexmon-capable Wi-Fi chips.

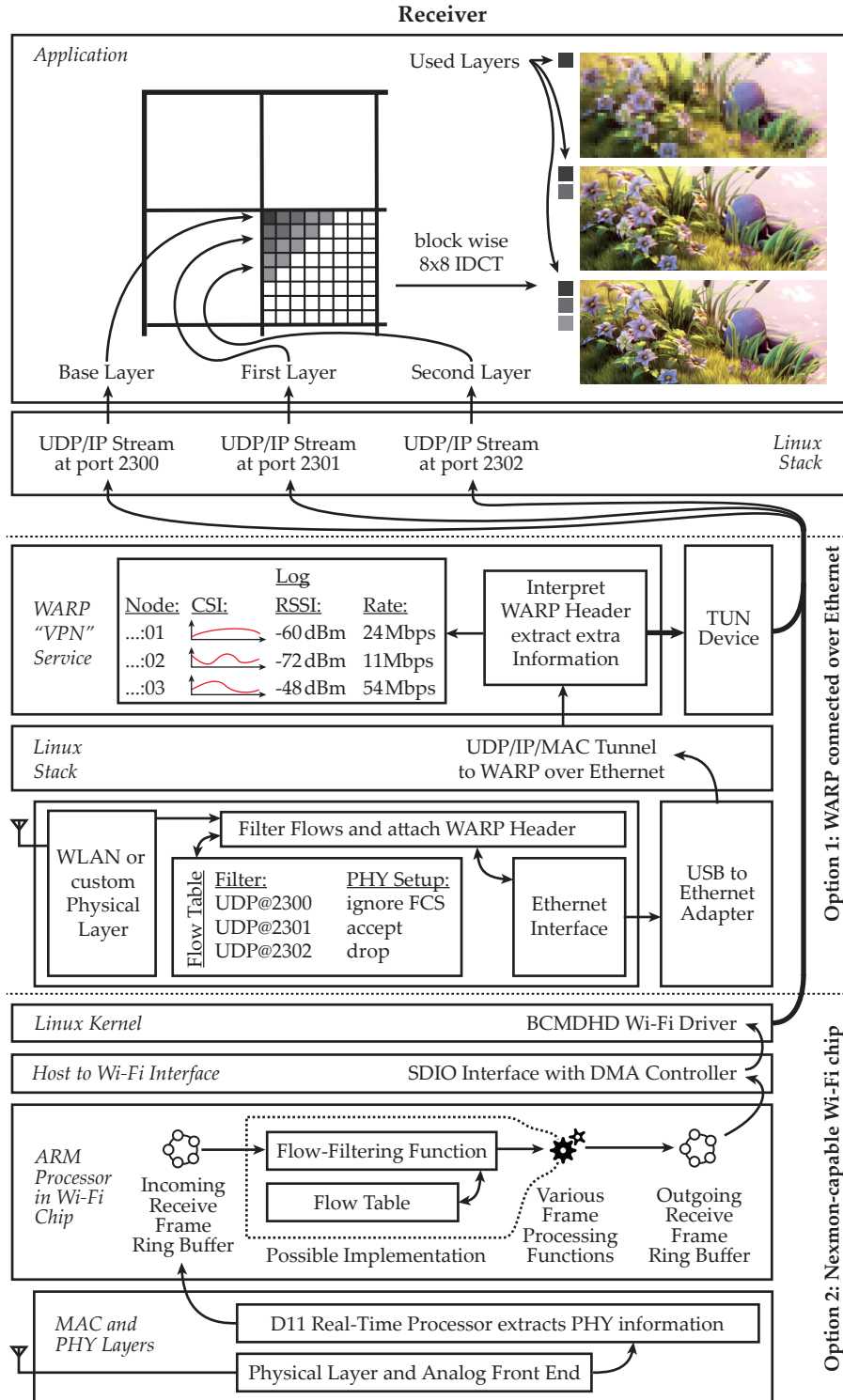


Figure 15: System overview that shows the components of our receiver. Frames are received either by Nexmon-capable Wi-Fi chips at the bottom or by WARP SDRs in the middle. Both can filter the incoming frames and extract additional information such as CSI. To interface the WARPs, we use USB-to-Ethernet adapters similar to the transmitter side and a VPN service to inject the received frames into the Linux stack. The Nexmon-based firmware patch does not require the VPN service. (based on [76])

layer. All other packets are handled by the existing network interfaces.

On the transmitter side, this WARP “VPN” Service takes all IP packets coming from the TUN device and attaches physical and data link layer headers to them. Those are similar to RadioTap headers used for injecting raw Wi-Fi frames with regular Wi-Fi chips. To decide which physical-layer features are required by an application, we use flow tables. These tables contain filters that match bits in packet headers as well as physical-layer settings that should be passed to the WARP. In our example the different SVC layers are transmitted using different UDP ports. Each of the three streams is considered a flow and can have unique requirements on the physical layer. For our Nexmon-based implementation, we could also rely on this VPN service to use the same system components as in the SDR-based implementation. However, we realized that the performance for processing frames can be increased by offloading the flow filters into the Wi-Fi firmware. As long as the Wi-Fi chip is the only interface we use for communication, getting rid of the VPN service is a suitable solution to optimize and simplify our implementation.

*Depending on the flow, the WARP “VPN” Service prepends additional MAC and PHY headers to outgoing frames that are interpreted in the SDR.*

*Using Nexmon we can offload the flow filters into the Wi-Fi firmware to increase the frame processing performance.*

#### 11.1.6 Interfacing SDRs from smartphones

We considered multiple interfaces for connecting with the WARP. To avoid interference during our wireless experiments, we decided against wireless interfaces. SDIO is not available on all smartphones. Directly interfacing the WARP using USB On-The-Go is not possible due to the lack of a USB interface at the WARP. However, we can use USB-to-Ethernet adapters to connect to the WARP’s 1 Gbps Ethernet ports. Over the Ethernet link, we tunnel our physical-layer frames to the WARP for transmission.

*Using a USB-to-Ethernet adapter seems to be the simplest way to connect WARPs to Android smartphones.*

#### 11.1.7 Offering enhanced features at the receiver

As long as we only intend to get more control over Wi-Fi transmissions and comply with the standard, we could use the smartphones’ regular Wi-Fi chips as receivers. Especially a Nexmon-capable Wi-Fi chip can provide interesting extensions at the receiver side. Flow filters could be used to drop frames not required by currently running applications or to decide which frames should be received even though the frame check sequence is wrong. A video application could cope with some bit errors and use at least the undamaged parts of a Wi-Fi frame. A firmware patch could also provide additional information about the reception quality to tune the streaming implementation. For example, by extracting channel state information (CSI) and feeding it back to the transmitter to apply beamsteering when broad-

*Nexmon-capable Wi-Fi chips can also implement flow filters at the receiver and extract additional information about the received frames.*



casting video frames. As explained in Chapter 8, Nexmon provides the CSI-extraction feature.

Nexmon also provides basic SDR capabilities to implement physical layers that differ from the Wi-Fi standard. However, in this case, frames need to be encoded and decoded in software. This significantly reduces the throughput and increases energy consumption. Hence, to evaluate new physical layers or non-standard compliant extensions, we need to use a full-fledged SDR at the receiver side that is able to decode custom physical layers in an FPGA. Of course, SDRs also allow to implement flow filters and to extract additional physical-layer information. After reception, selected frames are tunneled to the WARP “VPN” Service, that extracts the payload and passes IP packets on to the Linux kernel. The additional measurement information gets logged and can be used to enhance future transmissions as mentioned above.

*SDRs with FPGAs are able to decode custom wireless implementations in real-time.*

#### 11.1.8 Robust scalable-video transmission

As a use case for our application-centric physical layer, we chose video streaming. Videos have the advantage that their information can be encoded and compressed in various qualities. Decoders can cope with packet loss and bit errors which lead to image quality degradation that users may tolerate in real-time applications. To meet the requirements of heterogeneous end-devices as well as varying network speeds, scalable-video coding offers to encode video material into different layers. A low quality base layer is required by every user. To improve video playback, users can request additional layers. This principle is often illustrated by the SVC cube shown in Figure 16.

*We chose video streaming as example application, because it can cope with packet losses and bit errors.*

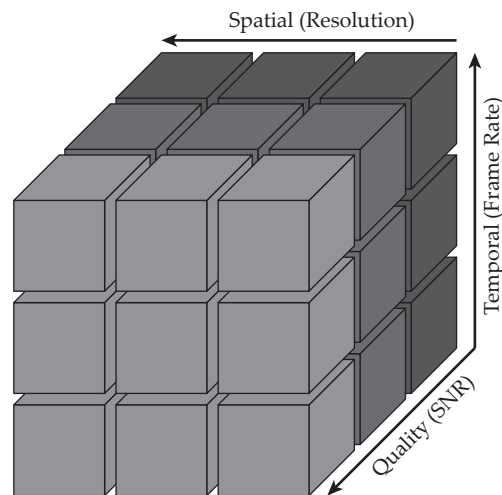


Figure 16: The H.264/SVC cube model to split a video into multiple quality, frame rate and resolution layers. We focus on varying the quality in our scalable-video codec implementation. (based on [76])

*Scalable-video codec implementations exist for H.264 and H.265, but they are very complex.*

*We created our own lightweight scalable-video codec based a Motion-JPEG-like compression.*

*Similar to JPEG, we use the discrete cosine transform to convert between spatial and frequency domain.*

*We use AX88179-based USB-to-Ethernet adapters to interface WARP SDRs.*

*We implement our codec in C to cope with the high computational load for encoding video frames.*

Currently, SVC extensions for existing video codecs such as H.264 and H.265 [86] exist in the form of H.264/SVC [81] and scalable high-efficient video coding (SHVC) [37]. However, those codecs have a high complexity, as they combine the idea of SVC with very efficient video compression. To give researchers a smooth entrance into the world of scalable video coding, we present a lightweight scalable video codec that is based on the idea of Motion JPEG, where each video frame is encoded separately using JPEG images. JPEG compression relies on the fact that high frequency components in a picture are less important for the human visual system than low frequency components. Our video codec uses this idea and only packs the lowest frequency components into the base layer, which is required by all receivers. Additional frequency components come with the first and second layer and are used to increase video quality. The result is illustrated in the upper right corner of Figure 15.

To get the frequency components, the video encoder applies the discrete cosine transform (DCT) to each  $8 \times 8$  block in each video frame on each of the three YCbCr color components. The receiver of the layered video stream converts the frequency components back to the spatial domain using the inverse discrete cosine transform (IDCT). The layers are required in the order base, first and second layer. If parts of a layer are missing, for example, due to frame loss on the physical layer, missing parts are replaced by parts from preceding frames.

## 11.2 IMPLEMENTATION

To implement the system described above, we used Nexus 5 smartphones. Their BCM4339 Wi-Fi chip is Nexmon-compatible and their USB port can be configured as USB host interface to connect AX88179-based USB-to-Ethernet adapters which interface WARP SDRs running the 802.11 reference design in ad hoc respectively IBSS mode.

### 11.2.1 Implementing the scalable-video codec

We implemented our scalable-video codec in C as this allows to optimize memory access much better than Java. To encode and decode videos in real-time, we have to calculate  $8 \times 8$  pixel DCT operations on all  $8 \times 8$  pixel block in each video frame. For a frame size of only  $800 \times 480$  pixels, we already need to calculate 6 000 of such DCTs. Without using optimized processor instructions for image processing, those operations already use up a significant amount of CPU time. However, by placing precalculated DCT coefficients efficiently in memory, we managed to optimize our C code sufficiently to encode videos in real-time. To read in video frames, we use OpenCV with builtin FFMPEG support. On all Android versions (at least between

4 and 6), we can read video frames from files. On Android 4, we can even use the build-in camera to encode videos. In any case, the video encoder produces three output streams containing frequency-domain components as illustrated in Figure 14 to represent our three quality layers. Color information is simply stored in an additional layer. We can either directly send each layer over UDP using separate UDP ports, or we can store the streams in a file and send them out in a second step. The first option is best suited for live demos of our app, while the second option helps to focus on networking aspects while running repeatable experiments.

The receiver works similarly. Video streams, can be either loaded from files or received over UDP. Then we apply the IDCT to generate  $8 \times 8$  pixel matrices containing the image information represented by the transmitted frequency components. We write the resulting images either directly to frame buffers that are displayed on the screen or into files. Again, the first option is used in live demos, while the second options allows to archive the received video information and analyze its quality separately. Overall, the implementation is independent of the underlying communication infrastructure and thereby reusable in other projects.

*Though our implemen-  
tation is not  
the most efficient, it  
transcodes videos in  
real-time and also  
works as a live demo.*

*The decoder directly  
writes video frames  
either to frame  
buffers used by the  
display or into files.*

### 11.2.2 Implementing the WARP “VPN” Service

The WarpVPNService class is used to handle the data processing between the WARP board and the Android device. When started, IP packets from applications running on the smartphone are captured with a TUN device. Access to the device is available without root privileges by implementing a VPN service. If desired, the service can also be bypassed by applications that do not require physical-layer control, by modifying the VPN’s subnet accordingly. The service is used to filter frames and attach WARP configuration headers according to flow table rules. During runtime, applications can send instructions to configure those flow filters using the Messenger interface of the service. This allows us to use a single service to map requirements of multiple applications running on a smartphone to physical-layer parameters. After attaching WARP configuration headers, the frames are packed into Ethernet frames and send over the USB-to-Ethernet adapter to the WARP. There, we interpret the header to set physical-layer parameters such as transmission power and modulation coding scheme accordingly.

*We use the VPN  
service to gain  
access to IP packets  
without requiring  
root privileges on an  
Android system.*

*Applications can  
reconfigure the flow  
filters in the VPN  
service during  
runtime.*

In the other direction, WARP nodes receive frames, add some meta data and send them encapsulated into Ethernet frames back to the smartphone. There, they are received by the VPN service. It extracts the IP packets and can log the additional meta information. After extraction, the IP packets are injected into the TUN device that hands them over to Linux’s networking stack. The VPN service is build in

*Received frames are passed from the WARP SDR to the VPN service and then injected into the networking stack.*

a way, that it can both be used by different applications and interface different physical-layer hardware. The WARP SDR is easily replaced by a USRP, an external Wi-Fi card or an LTE interface. It is simply a frame processing element that separates flows and attaches a set of settings for their transmission depending on the requirements set for each flow.

### 11.2.3 A Nexmon-based implementation using internal Wi-Fi chips

*In a first step, we connected the WARP VPN Service to a Nexmon-based firmware with monitor mode and injection support.*

Our Nexmon-based implementation is currently a work in progress. Florentin Putz first started to modify the WARP VPN Service to interface a Nexmon firmware that supports monitor mode and frame injection. Instead of exchanging frames with the WARP over Ethernet, it directly interfaces the internal Wi-Fi chip. To set transmit parameters, we can prepend the transmitted frames with RadioTap headers. In this case, we would first have to pack the IP packets into Wi-Fi frames in the VPN service and then send them to the Wi-Fi chip. While this approach works, it is unaware of the current state of the Wi-Fi connection managed by the Wi-Fi firmware. A better approach would be to simply pass Ethernet frames to the Wi-Fi firmware. The firmware expects those anyways and repacks them into Wi-Fi frames internally. We just have to pass the physical-layer parameters that have to be used to the firmware.

*It is more efficient to ignore the VPN service and instead embed its functionality directly into the Wi-Fi firmware.*

In the case, where the Wi-Fi chip should handle all outgoing traffic anyways, it would be much simpler to bypass the WARP VPN Service and send all outgoing traffic directly to the Wi-Fi interface as it is done by default. To manage flows, we simply move the flow filters from the WARP VPN Service into the firmware. Here, we can inspect the payload of all outgoing frames after attaching Wi-Fi headers. If frames are detected as part of a flow whose physical-layer settings should be controlled, we can directly set the physical-layer parameters in the `d11txhdr` structure prepended to each outgoing frame, as described in Section 6.2.2.

*Flow filter implementations can range from hardcoded into the firmware binary to reprogrammable during runtime using position-independent code.*

We define flow filters and their effect in frame-processing functions. There are various ways to implement them in the firmware. For static experiments, we can hardcode those filters into the firmware binary and reload the firmware whenever our experimental setup changes. To enhance this implementation, we can implement flow filters that can be updated. We keep a list of conditions with corresponding physical-layer settings. The code to check the conditions with the processed frames and apply the settings is hardcoded, but the list can be updated using an `ioctl`, as explained in Section 6.2.13. This approach already reaches the flexibility of the WARP VPN Service. To further enhance it, we can side-load the complete frame-processing function during runtime. To this end, we define a clean interface for passing frames through the function. As side-loaded functions have

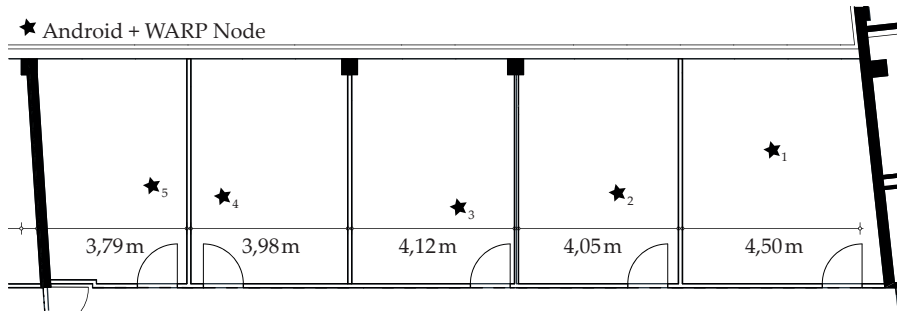


Figure 17: Experimental setup in our office environment. All phones are hanging on movable hat stands in a height of roughly 170 cm. Positions look random as the devices are placed around the tables in our offices. (based on [76])

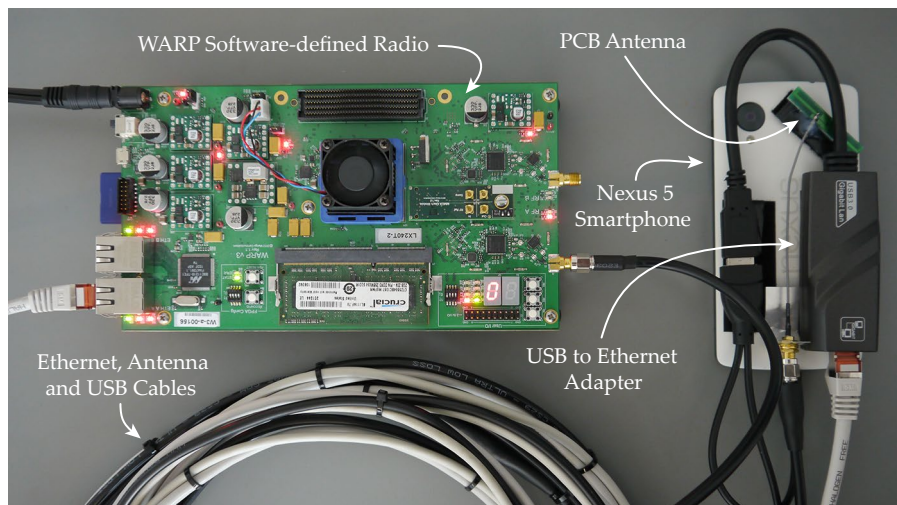


Figure 18: To each Android node we attach a USB-to-Ethernet adapter with a 5 m long cable to the WARP and a power supply. Using a low-loss CS29 cable, we feed back the WARP's radio frequency (RF) output to a PCB antenna (3 dBi gain) that is fixed to the smartphone. (based on [76])

full access to the Wi-Fi chip, we have to take security precautions as presented in Section 6.1.4. Having a set of side-loadable functions that are digitally signed can limit security issues and keep the size of the main firmware binary small while being able to load frame-processing functions on demand. Those processing functions are also not limited to filtering flows and defining physical-layer settings. We could also define that some data should be transmitted by using the SDR features of the Wi-Fi chip described in Chapter 9 to, for example, perform cross-technology communication.

*SDR features could be employed for cross-technology communication.*

## 11.3 EVALUATION

*We use five Nexus 5 smartphones connected to WARP SDRs whose antennas are attached to the backs of the smartphones.*

*Node 1 broadcasted a three layered video stream to the other smartphones in ad hoc mode.*

To evaluate the feasibility of our solution, we set up a testbed in our office environment as illustrated in Figure 17. We used five smartphones that are equipped with WARP software-defined radios (SDRs), as illustrated in Figure 18. Each WARP SDR is connected to a USB-to-Ethernet adapter that is connected to the USB port of the smartphones. To transmit and receive signals from the smartphone's location, we attached PCB antennas to the backs of the smartphones and connected them to the antenna ports of the WARP SDRs. To reduce interference by other Wi-Fi users, we performed experiments at night and selected channel 14 in the 2.4 GHz band with 20 MHz bandwidth. In Germany this channel is reserved for research purposes.

During our experiments, we used Node 1 as the transmitter that generated a layered SVC video stream. We transmitted each of the three layers on a separate UDP port resulting in three different flows. Depending on the experiment, we varied the used modulation scheme, the amount of forward error correction (FEC) and the transmit power for each flow. The transmitter fragmented all video frames into 5 (base layer), 23 (first layer) or 41 (second layer) Wi-Fi frames (with 1348 byte payload) that we broadcasted in Wi-Fi ad hoc mode without retransmissions. Nodes 2 to 5 were receivers, on which we measured the number of received Wi-Fi frames and the quality of the video playback in terms of structural similarity (SSIM).

## 11.3.1 Experiment definition

Each of our experiments lasted 20 seconds (480 video frames at 24 frames per second). We measured video qualities on a per-frame basis and frame reception rates per second and plotted the average including 99 percent confidence intervals. As we do not focus on

Table 2: 802.11g data rates achievable on WARP

PHY	Gross Rate	Achievable Rate	Modulation	FEC	Payload	Air Time
	6 Mbps	5.3 Mbps	BPSK	1/2		1.979 ms
	9 Mbps	7.6 Mbps	BPSK	3/4		1.198 ms
	12 Mbps	9.8 Mbps	QPSK	1/2		0.899 ms
	18 Mbps	13.6 Mbps	QPSK	3/4		0.599 ms
	24 Mbps	17.0 Mbps	16-QAM	1/2		0.449 ms
	36 Mbps	22.2 Mbps	16-QAM	3/4		0.300 ms
	48 Mbps	26.4 Mbps	64-QAM	2/3		0.225 ms
	54 Mbps	28.1 Mbps	64-QAM	3/4		0.200 ms

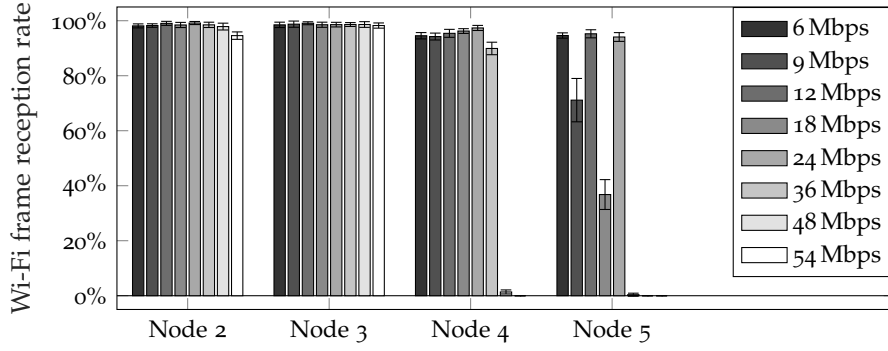


Figure 19: Wi-Fi frame reception rates of base layer frames transmitted at different gross data rates at 10 dBm transmit power. (based on [76])

video compression, the bit rates per frame are comparably high. However, this allows us to test our system under load requirements for high-definition videos. For our frame size of  $800 \times 480$  pixels and  $8 \times 8$  pixel block size, we required 17.280 Mbps for all three grayscale video layers (1.152 Mbps for the base, 5.760 Mbps for the first and 10.368 Mbps for the second layer).

According to [89] and considering 1348 payload bytes per Wi-Fi frame, the achievable data rates lie below the gross data rates (see Table 2). Hence, we needed to choose a data rate of 36 Mbps to have sufficient throughput for our 17.280 Mbps video stream. This 36 Mbps setting acts also as the baseline for further experiments. To analyze the video quality as well as energy efficiency, we defined two sets of experiments: (1) We used a transmit power of 10 dBm and kept the first and second layer's bit rate fixed at 36 Mbps and changed the rates of the base layer to all rates given in Table 2. (2) We fixed the rates of the base, first and second layers to 6, 24 and 48 Mbps and changed the transmit power between  $-12$  dBm to 18 dBm in steps of 3 dB.

### 11.3.2 Evaluation of transmit rate variations

In the following, we present our results. In Figure 19, we show the rates of successfully received Wi-Fi frames. Those have a correct frame check sequence (FCS). With the given transmit power, Nodes 2 and 3 have almost no packet loss, while Node 4 only receives packets up to a rate of 36 Mbps. Node 5 is the most distant node from the transmitter and receives well at rates of 6, 12 and 24 Mbps. Higher rates result in almost no reception, while rates of 9 and 18 Mbps endure higher frame loss than their neighboring higher rates. This effect is most likely due to the fact that at 9 and 18 Mbps each *three* data bits are protected by one FEC bit, while at 6, 12 and 24 Mbps, *each* data bit is protected by one FEC bit. Under the assumption that the reference

*Due to the low but lossy compression of our simple scalable-video codec, we generate video streams with high data rates.*

*In our experiments, we either fix the transmit power and vary the modulation coding scheme (MCS) or fix the MCS and vary the transmit power.*

*At higher distances to the transmitter, the reception rates drop for less robust modulation coding schemes.*

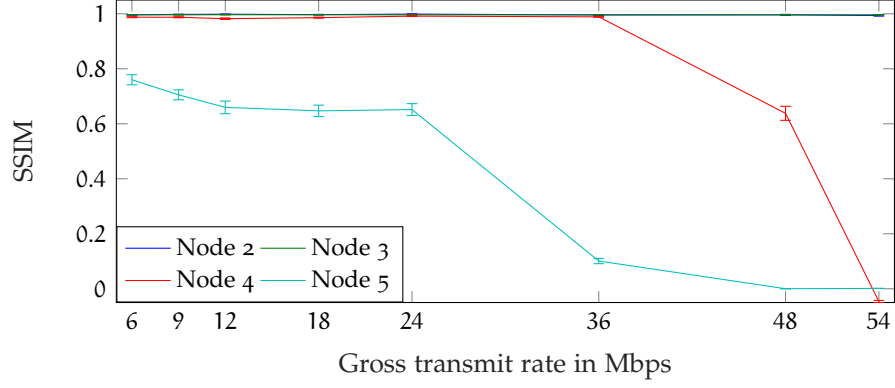


Figure 20: Video qualities measured as SSIM when the transmit power is fixed at 10 dBm and the transmit rate of the base layer changes while keeping the first and second layer fixed at 36 Mbps. (based on [76])

design implementation for the WARP is correct, we conclude that gaining throughput by increasing modulation orders is more efficient than reducing FEC.

Regarding video quality (see Figure 20), we observe that Nodes 2, 3 and 4 have SSIM values close to one, which means that they have very good video quality. Compared to Node 5, they receive the first and second layers in addition to the base layer as they have low frame loss rates at 36 Mbps (at which the higher video layers are transmitted). Node 5 does not receive the higher layers, therefore, it only achieves an acceptable video quality around 0.7. As soon as we increase the base layer's transmit rate above the point of low frame loss rates, we encounter very bad video qualities below 0.2, even though the higher layers might still be receivable. We conclude that video quality is highly correlated with frame loss rates and that it is essential to ensure a good reception of the base layer which alone gives acceptable video quality.

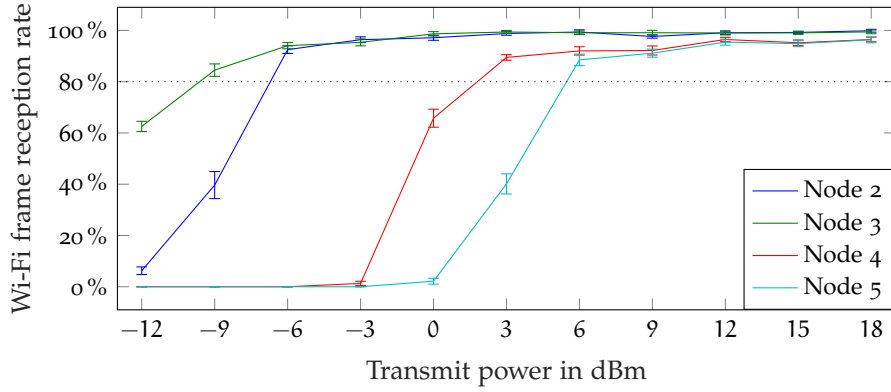
*As long as the base layer is correctly received, we get at least acceptable video qualities which underlines the requirements for robust transmissions.*

### 11.3.3 Evaluation of transmit power variations

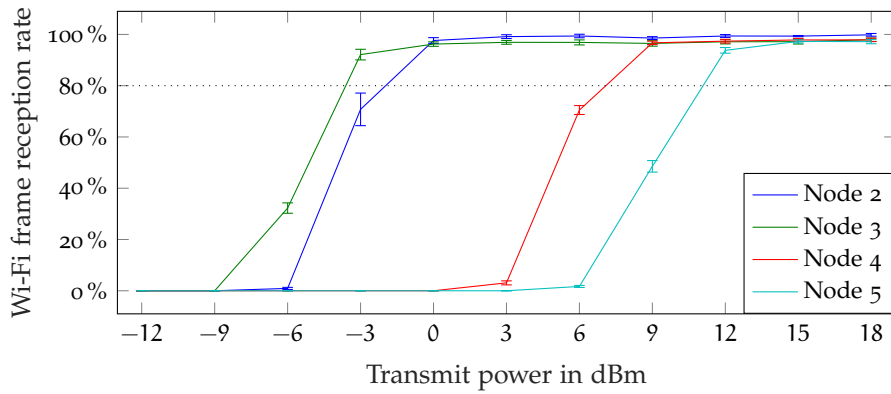
In our second set of experiments we analyzed the influence of transmit power on video reception. We selected a robust transmit rate of 6 Mbps for the base layer and less robust rates of 24 Mbps and 48 Mbps for the first and second layers. Figure 21 illustrates that the higher the transmit rate, the higher the transmit power requirements to successfully receive the Wi-Fi frames. However, if the transmit power gets too high, transmission at high transmit rates becomes impossible, which is most likely due to the errors introduced by high and therefore non-linear amplification. The effect is observable in Figure 21(c) at a transmit power of 18 dBm. We also observe that nodes closer to the transmitter are able to receive at lower powers

*The higher the transmit power, the better high throughput transmissions can be received as long as too high amplification does not degrade the signal qualities.*

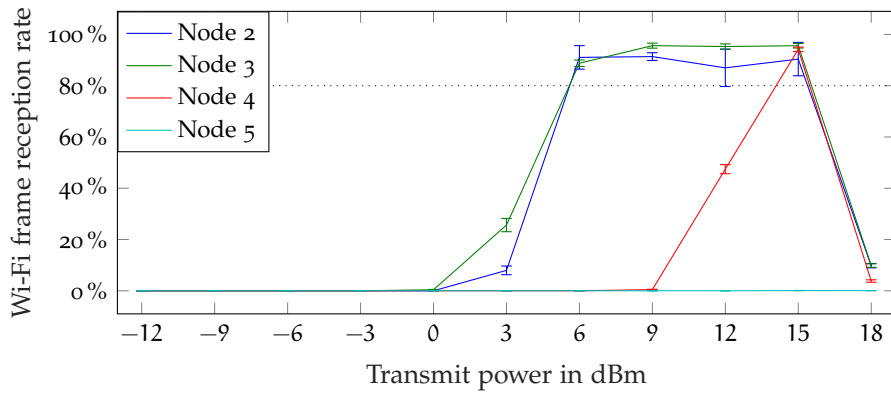




(a) Base layer at 6 Mbps



(b) First layer at 24 Mbps



(c) Second layer at 48 Mbps

Figure 21: Wi-Fi frame reception rates when keeping the base, first and second layer transmit rates fixed at 6, 24 and 48 Mbps, while varying the transmit powers. (based on [76])

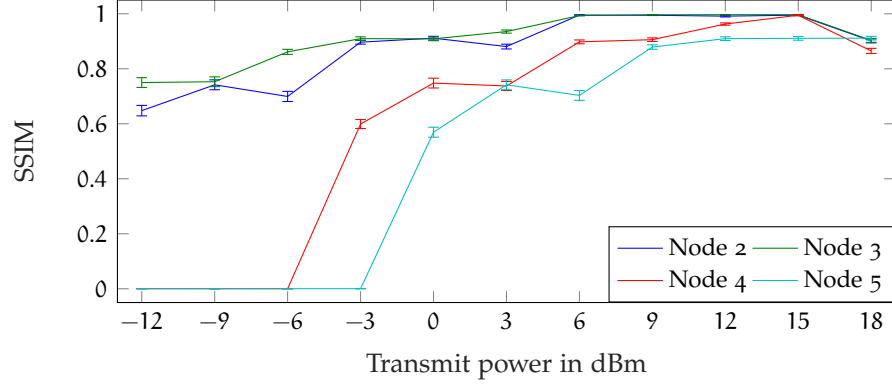


Figure 22: Video qualities measured as SSIM when keeping the base, first and second layer transmit rates fixed at 6, 24 and 48 Mbps, while varying the transmit powers. (based on [76])

than nodes that are further away. Only at low transmit powers Node 3 has a better reception than Node 2. This is either due to different WARP boards whose receiver sensitivities can vary from board to board, or it is due to the direction the smartphones are facing in the room. As the antenna of Node 2 faces away from the transmitter it might receive noisier signals than Node 3.

Video qualities illustrated in Figure 22 reflect the observations of the frame reception rates. At low transmit powers Nodes 2 and 3 already get medium video quality that increases to very good quality as soon as transmit powers reach 6 dBm so that the second layer is receivable. Video reception at Nodes 4 and 5 start at  $-3$ , respectively  $-6$  dBm with acceptable quality.

*Video qualities correlate with the ability to receive frames of different quality layers.*

#### 11.4 DISCUSSION

Our experiments show that allowing applications to change physical-layer parameters by using flow tables is practical. Using a VPN service to implement an abstraction layer has multiple benefits. First, existing IP-based applications can profit from physical-layer optimizations. Second, the service abstracts from complex physical-layer parameters by translating application requirements to physical-layer settings that can vary between wireless technologies. Third, the flow-based approach avoids an over-optimization on a per-frame basis. Instead, applications need to set requirements only once per flow and our service is responsible to choose and adapt physical-layer settings over time. Nevertheless, running the VPN service also generates an overhead that can be avoided by offloading the flow filtering functions directly into the communication hardware. To this end, we proposed to modify the Wi-Fi firmware of a Nexmon-capable Wi-Fi chip.

In our evaluation, we demonstrated that our solution is suitable to implement and optimize application-centric physical layers that

*The implementation based on a VPN service is transparent to IP-based communication, it abstracts complex physical-layer settings by processing requirements and allows working with flows.*

are aware of application requirements. For video streaming, our results show that adjusting transmit power and transmit rates already allows optimizations that help to meet video quality requirements at receivers. Our solution is further capable of adjusting more physical-layer parameters or even exchange complete physical-layer implementations on the SDRs, depending on application requirements.

## 11.5 FUTURE WORK

While the current implementation can be considered as a proof-of-concept, various extensions are possible. As mentioned above in Section 11.2.3, connecting our application with Nexmon-capable Wi-Fi chips is currently a work in progress. After this extension, we plan to replace our custom scalable-video codec by a standard codec implementation. Either supporting scalable-video coding or other streaming approaches such as DASH that support adjusting video quality parameters during playback. Other applications that might benefit from physical-layer control such as massive multi-player gaming or emergency applications are also worth investigating.

Additionally, we did not use the full potential of software-defined radios in our implementation. By switching between physical-layer standards or using the SDR capabilities of Wi-Fi chips to perform cross-technology communication (CTC), one could further enhance the interoperability between various wireless standards, chose the best standard for the type of data that should be transmitted or investigate how upcoming standard perform as physical layer in a live video streaming scenario. By reverse engineering the operation of the Bluetooth components build into Broadcom's Wi-Fi chips, we could at least switch between Wi-Fi and Bluetooth communication on demand, for example to enhance energy consumption by only activating the Wi-Fi receiver as soon as a video transmitter is detected using Bluetooth.

## 11.6 RELATED WORK

Besides gaining lower-layer access on commercial off-the-shelf hardware, which is offered by Nexmon, this work especially relates to three main areas, flow-based networking approaches with focus on the wireless domain, and scalable-video-coding schemes.

To bundle and manage network frames that logically belong together, we rely on the concept of flows, an idea introduced in the 1960s [27]. Flows can either be extracted from header information, for example ports and addresses, or they can be explicitly defined by using flow labels as in IPv6 [71]. OpenFlow [64] is a project implementing flows in network hardware to allow controllers direct access to flow tables, for example, in switches to control how data is

*Just by adjusting transmit power and modulation coding schemes, we can optimize video transmissions.*

*We are working on a Nexmon-based implementation with support for standard video codecs and other applications.*

*Switching between physical layers allows to chose the best implementation for the application at hand and can combine the benefits of different communication standards.*

*Our work relates to flow-based communication and scalable-video coding.*

*The concept of flows exists since the 1960s and is used by the SDN applications OpenFlow and OpenRoads.*

forwarded. It implements the concept of software-defined networking (SDN). OpenRoads or OpenFlow Wireless [99] is the corresponding project for wireless systems. Its major purpose is to allow handovers between wireless technologies and to manage wireless networks. OpenRadio [5] is a more application-centric solution that aims at developing a platform that supports multiple physical layers that can be adjusted to the needs of an application. However, it misses a complete implementation and extensive evaluation.

*Flows are commonly used for handling video streams in a network.*

Video streaming is already extensively evaluated in various works. The authors of [45] focus on flow-based control for video streaming services and significantly enhance the quality of experience (QoE) of video on demand (VoD) services like YouTube. Therefore, they rely on deep packet inspection (DPI) and application metrics. There are also works like [46] that focus on optimizing physical-layer transmission for video applications in multi-antenna systems. The physical layer knows which bits are more important for a successful video transmission and can adjust according to channel conditions. Last but not least, SVC is another solution to cope with fluctuating network throughput by encoding video into multiple quality layers that have to arrive at receivers with different priorities. This makes SVC streams ideal applications for SDN. In [53], the authors create flow based routing protocols to cope with varying network conditions as well as playback device properties. In practise, SVC extensions exist for the H.264 and H.265 codecs [37, 81]. However, the support for mobile playback devices is still very limited. The authors of [55] offer at least a software-based player for Android devices. In this work, we neither intend to reinvent scalable-video codecs, nor do we want to compete against existing solutions. Our intention is to have a lightweight video codec based on a simple implementation that runs smoothly on mobile devices and is easily extensible so that we can combine it with the concept of flows as well as physical-layer optimizations.

*SVC is a well suited application for software-defined networks.*

## 11.7 CONCLUSION

*Our solution allows mobile applications to modify physical-layer parameters on a flow basis by defining requirements for the transmissions.*

In this chapter, we presented a solution that allows mobile applications to take advantage of physical-layer properties when transmitting data. Our framework relies on the concept of flows to avoid over-optimizing the physical layer on a per-frame basis. For each flow, an application can define requirements on the physical transmission such as the use of robust modulation schemes and high power transmissions if a flow should be received by as many receivers as possible, or the combination of high throughput and low transmit power to only serve nodes in close proximity. To abstract from the complexity of the physical layer and to support multiple applications per device, we introduce an intermediary layer that handles applica-

tion requirements and sets corresponding physical-layer parameters according to flow tables. By implementing our solution as VPN service on Android, it becomes transparent to existing applications that rely on IP based communication.

As an example application to evaluate our solution's performance, we chose scalable-video coding on Android smartphones. We designed and implemented a lightweight scalable-video codec that focuses only on offering multiple quality layers that can be transmitted over separate flows. To be able to change physical-layer parameters on smartphones and to have flexibility for future research, we used software-defined radios that currently implement the 802.11g standard in an easily extensible way. To simplify the setup, we also proposed to replace the SDR by a Nexmon-capable Wi-Fi chip that has control over various physical-layer parameters on off-the-shelf devices. In our evaluation, we analyzed the effect of choosing different modulation coding schemes and transmit powers on frame reception rates and video quality. In the next chapter, we continue with more security focused applications.

*Our example application is a custom implementation of a lightweight scalable-video codec.*

*Either SDRs or Nexmon-capable Wi-Fi chips realize the requirements set by an application.*

## 11.8 MY CONTRIBUTION AND ACKNOWLEDGEMENTS

While looking into research topics related to the *Multi-Mechanisms Adaptation for the Future Internet (MAKI)* project, I came up with the idea to optimize the video-streaming scenario, often used in this project, by providing means that let applications control physical-layer properties. To this end, I developed the scalable-video codec, the interface to the WARP and the VPN service. When Nexmon entered a production state, I had the idea to replace the WARP SDR by a Nexmon-capable Wi-Fi chip. The adaptation is currently in process and Florentin Putz is working on its implementation. I thank him for his eager work and the interest in the continuation of this project. Additionally, I thank Denny Stohr for collaborating on the first version of the system with me.

*This work is based on a collaboration within the MAKI project.*



Wireless radio communication jammers have been around for decades. They are used for strategic advantages, hindering an opposing party from exchanging information, for example, in a military conflict or situations where remote trigger signals for explosive devices need to be suppressed. They are also used to protect vulnerable legacy systems from malicious communication [12, 14, 31, 61, 83, 92, 95], for example, pace makers that can be wirelessly reprogrammed without encryption and no authentication. Reactively jamming all unauthorized communication with those devices can be a life saver and protect a patient's privacy [31, 95].

Besides using jammers for friendly or public safety applications, radio jammers are also subject to abuse. Whoever owns a jammer for GSM and LTE bands can block cellular communication and in doing so also hinder victims in distress situations from using phones to make 9-1-1 calls to call for help. People tracked by the government with GPS anklets may use jammers to leave their allowed living zone but may additionally disturb other GPS applications in their vicinity. While those applications are definitely illegal in most countries of the world (see [20]), it is important to understand what malicious attackers can achieve. Existing radio communication hardware carried around by ordinary civilians can participate in attacks that we need to prevent.

Smartphones, for instance, may be the most widespread radio transmission enabled devices. They are carried around by billions of people every day and are densely distributed in metropolitan areas where people constantly communicate wirelessly. A malicious attacker overtaking only a small fraction of these devices could create a network of densely distributed highly capable radio jammers that could trigger a wide-spread denial-of-service attack against wireless communication. In addition, reactive jamming would allow to create a mesh network of cooperative distributed jamming nodes that could selectively jam any other communication while keeping an open control channel for their coordination.

In this chapter, we investigate the feasibility of smartphone-based wireless jammers by implementing proof-of-concept firmwares for the Nexus 5 smartphone. The goals are always friendly-jamming applications to either hinder nodes from transmitting non-compliant Wi-Fi signals, or to protect industrial resources by using large numbers of employee smartphones to defend against attacks on legacy hardware. To this end, we implement a smartphone-based reactive

*Friendly jammers are used to protect unencrypted and unauthenticated wireless applications against illegal signal injections and information leakage.*

*As dual-use applications, jammers may also be abused to hinder the placement of distress calls or hamper positioning systems.*

*Attacks may turn omni-present communication devices such as smartphones into a decentralized network of mobile jammers.*

*We use Nexmon to create firmware patches that implement a reactive jammer in the Wi-Fi chip of a Nexus 5 smartphone.*

*To avoid suppressing all communication of a node, we introduce the concept of an acknowledging jammer with adaptive power-control abilities.*

*For our implementation, we use the D11 core to analyze incoming frames in real-time and, then, trigger a transmission of a jamming waveform stored in the sample-play buffer.*

*We are the first to present smartphone based jamming that even targets 802.11ac transmissions.*

*Analysing and modifying ucode to transmit arbitrary signals in time was challenging.*

jammer for Wi-Fi systems that can jam all receivable rates supported by Nexus 5 smartphones (e.g., 80 MHz SISO 802.11ac frames). During our experiments, we realized that reactive Wi-Fi jammers might target only single communication flows, but always hamper the complete communication of a specific device as the physical layer in the transmitter does not differentiate to which flow a frame belongs. As a solution for this problem, we enhance the reactive jammer by sending acknowledgements to the frame transmitter. This avoids retransmissions and the blockage of non-targeted traffic. To optimize the runtime of smartphone-based jammers, we further improve our jammer by using an adaptive power-control algorithm to adjust the transmission power depending on the jamming success. For each of the three jamming approaches, we evaluate the jamming performance and the energy consumption. All of our experiments are designed with reproducibility in mind. Even though, our jammer would allow continuous-tone jamming, we solely focus on reactive jammers as they are required for friendly jamming applications.

To implement the reactive jammer, we add a simple parser to the programmable state machine running in the D11 core (see Section 4.2.2 and Section 4.4 for background information and Section 6.2.9 and Section 6.2.14 for implementation information). It inspects a frame's header while the baseband is still receiving the frame data. If a jamming condition matches (e.g., the destination UDP port), we start a new transmission that interferes with the target frame at the receiver. While Berger et al. [8, 9] request the baseband to immediately transmit an 802.11 frame, we force the chip to transmit an arbitrary waveform that we previously stored in the sample-play buffer. It contains up to 512 complex samples that are fed into the DACs for a configurable number of times. We can, hence, control the spectral shape of the jamming signal, its duration and power. With respect to [8, 9], our approach is more flexible as it allows a finer customization of the jamming parameters. For instance, we can focus the jamming energy on selected carriers like the pilots that are used at the receiver for signal equalising [17, 36, 82]. All of this is actually possible thanks to our discovery of new chip features, that were not available in the "old" 802.11g chips used in [8, 9]. For the same reason, we are also reporting, for the first time, the performance of reactive jamming with different settings of the channel bandwidth (40 MHz and 80 MHz) introduced by the 802.11ac standard.

Developing our three jamming applications, we faced the following challenges: (1) We had to reverse engineer the ucode of the real-time processor, (2) we had to understand how arbitrary signal transmissions work, (3) we had to trigger those transmissions in time from within the ucode, and (4) we had to understand how to unlock arbitrary modulation rates during transmission.



In the following, we first present the high-level design of our three jammers in Section 12.1, continue with a detailed description of their implementation in Section 12.2 and an experimental evaluation in Section 12.3. In Section 12.4, we discuss smartphone-based jamming, present related work in Section 12.5 and future work in Section 12.6. Then, we conclude in Section 12.7.

*After a high-level design follows a detailed implementation and an experimental evaluation.*

## 12.1 DESIGN

In this section, we present the design of our three jammers. As continuous jammers are too destructive for friendly-jamming scenarios, we only build reactive jammers. The first type—called reactive jammer (see Section 12.1.1)—transmits pilot tones whenever a jamming condition matches. The second type, the acknowledging jammer (see Section 12.1.2), is a novel extension to reactive jammers and sends back acknowledgements to the transmitter of a jammed frame. This avoids blocking non-targeted streams queued for transmission, as well as fallback to more robust modulation and coding schemes (MCSs). To reduce power consumption of our second jammer, we introduce the adaptive power-control jammer (see Section 12.1.3) as third type. It adjusts its transmission power according to an observed jamming success indication. We also present how to generate jamming signals that fit into the sample-play buffer (see Section 12.1.4) and describe how to amplify them in the analog front end of the Wi-Fi chip (see Section 12.1.5). At the end, we perform some basic energy consumption measurements to get a feeling on how much power is consumed for transmitting signals (see Section 12.1.6).

*We describe the design of the reactive jammer, the acknowledging jammer and the adaptive power-control jammer.*

### 12.1.1 Reactive jammer

A sophisticated jamming approach is a reactive jammer that only jams if a communication is detected and a jamming condition matches. Our reactive jammer is similar to the one presented by Berger et al. in [8, 9]. However, instead of using Broadcom’s SoftMAC chips of wireless routers, we run our jammer in the FullMAC Wi-Fi chip of the Nexus 5 smartphone. Besides an increased mobility, it also supports 802.11ac to analyze more frame types and allows the transmission of arbitrary waveforms read from IQ buffers.

*Our reactive jammer is similar to the one implemented by Berger et al. on SoftMAC Wi-Fi chips, but supports the transmission of arbitrary waveforms.*

The jammer should work as follows. Whenever a Wi-Fi frame is being detected, the programmable state machine implemented in the D11 core waits until enough bytes are received to check our jammer conditions. In our experiments, we compare the UDP port numbers found in unencrypted Wi-Fi frames against a target port number. In case the frame should be jammed, the D11 core switches from reception to transmission and triggers the playback of the sample-play buffer that contains the samples of our jamming signal, as described

*If the D11 core decides that a frame matches a jamming condition, it starts the transmissions of a jamming signal.*

*We transmit jamming signals from the sample-play buffer that we discovered before transmission capabilities from Template RAM.*

in Section 4.2.2. We chose to store the jamming signal in the sample-play buffer, as transmission from the much larger Template RAM was not discovered before finishing the jamming project. Additionally, we only need simple waveforms to successfully jam Wi-Fi frames, that easily fit into the sample-play buffer. After finishing the transmission of the jamming waveform, we have to reset the physical layer into receive mode and wait for new frames to arrive. In case a frame should not be jammed, we proceed as regular Wi-Fi stations do. Either drop the frame or receive it and forward it to the ARM processor for further processing.

### 12.1.2 Acknowledging jammer

*While the reactive jammer can target single flows coming from a node, it always throttles any communication of this node.*

During our pre-evaluation of Wi-Fi jamming scenarios, we realized that nodes sending multiple parallel data flows throttle all of their transmission, if only one flow gets jammed. This is due to the fact, that the MAC layer in the Wi-Fi chip does not differentiate between flows. Those flows could be differentiated by protocols and ports on the transport layer. Communication nodes may run different services over different flows. For example, a video streaming application sends data on a different flow than an audio conferencing system. In case the jammer wants to block only selected network services but ensure that other services continue to run on the jammed nodes, an extension to our reactive jammer is required.

*Data can only continue to flow, when the transmitter of jammed frames thinks that its frame was correctly received.*

Transmissions throttle as the Wi-Fi chip expects acknowledgements for every transmitted frame. If those are missing, it performs retransmission with an ever growing backoff time until a retransmission counter runs out. Hence, all other queued outgoing frames need to wait. To circumvent this problem, without changing the operation of regular Wi-Fi nodes, we need to fake acknowledgements with our jammer. To this end, we need to schedule a new acknowledgement in the transmit engine of the D11 core. Even though the jamming signal is transmitted over the frame that is currently being received, the state machine of the D11 core anyways reacts at the end of a reception to evaluate the correctness of an incoming frame. We use this event to schedule the transmission of the fake acknowledgement with the correct timing expected by the targeted frame's transmitter.

### 12.1.3 Adaptive power-control jammer

*We can optimize the jammer's battery life by minimizing the jamming power.*

The jammers presented above transmit at a fixed power that we configure before the experiments. We usually choose a high power to ensure successful jamming results. In many scenarios, however, this value exceeds the actually required minimum power to destroy frames at the receivers. This choice is not only energy inefficient but may also

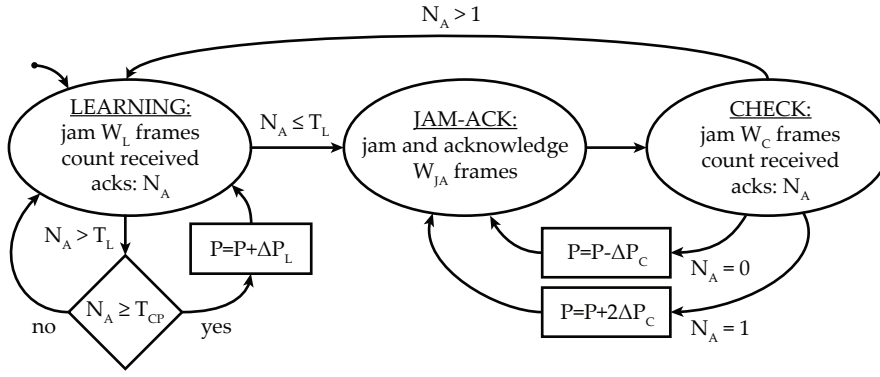


Figure 23: State machine of the adaptive power-control jammer. (based on [74])

disturb neighbouring communications. To address these two issues, we improve our jammers by an adaptive power-control algorithm.

We illustrate this algorithm in the form of a state machine in Figure 23. It consists of the three states “LEARNING”, “JAM-ACK” and “CHECK”. In the initial state “LEARNING”, we jam every target frame in a window of  $W_L$  received frames and count how many acknowledgement frames  $N_A$  come from the receiver. If  $N_A$  exceeds the targeted minimum  $T_L$ , a new learning window starts. If  $N_A$  is also larger than  $T_{CP}$ , the transmission power is increased by  $\Delta P_L$ . If  $N_A$  is below or equal to the targeted minimum  $T_L$ , we enter the “JAM-ACK” state.

In this state, we jam and acknowledge a window of  $W_{JA}$  frames. If the power determined before is sufficient, we keep jamming successfully while sending correctly forged acknowledgements to the transmitter. As a consequence, the transmitter keeps a high data rate and a very short contention window that avoids throttling additional data streams. At the end of each “JAM-ACK” window, we enter the “CHECK” state.

Here, we test whether a new learning phase is required by jamming a window of  $W_C$  frames without acknowledging them. Instead, we again count how many acknowledgements  $N_A$  come back. In case,  $N_A$  is greater than one, a new learning phase is started. Otherwise, we reenter the “JAM-ACK” state and either reduce the jamming power by  $\Delta P_C$  if no acknowledgements were received (jamming was perfect), or we increase the jamming power by  $2 \cdot \Delta P_C$  if exactly one acknowledgement was received (meaning we are directly at the edge of the minimum required jamming power). This state machine is by far not optimal and we did not consider convergence properties of the algorithm. Instead, we only intent it to be a proof-of-concept for demonstrating that it works in a practical setup.

*In the “LEARNING” state, we jam frames and check whether acknowledgements are transmitted by the receiver.*

*In the “JAM-ACK” state we are blind to acknowledgements as we send our own.*

*Regularly, we test in the “CHECK” state whether our jamming power should be increased or reduced.*

*The state machine has optimization potential.*

#### 12.1.4 Jamming signal generation

*We use the IDFT to generate cyclically repeatable waveforms that fit into the sample-play buffer.*

For all three kinds of jammers, we need to generate a jamming signal and store it in the sample-play buffer. The latter is limited to 512 samples that can be played back in a loop. To avoid discontinuities at the end of a buffer, we only generate tones on frequencies whose periods completely fit into the buffer. To this end, we use an inverse discrete Fourier transform (IDFT). While an inverse fast Fourier transform (IFFT) is optimized for sizes that equal powers of two (1, 2, 4, ..., 512), the IDFT works with arbitrary numbers of samples to operate on. This allows us to generate cyclic buffer contents with arbitrary subcarrier spacing. For example, to generate tones with a spacing of 1 MHz on a channel with 80 MHz bandwidth that is sampled at 160 MSps, we use a 160 samples long IDFT and can define an amplitude and a phase for each subcarrier.

*Choosing appropriate phases helps to reduce the PAPR of a signal which simplifies amplification.*

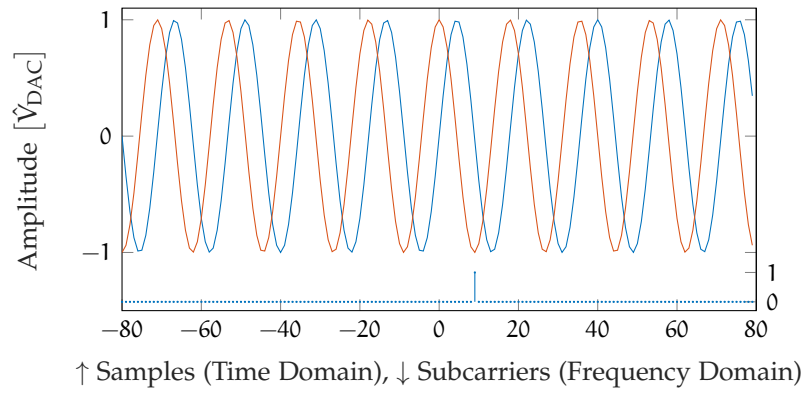
In the jamming scenario, the phase of the tones is of minor importance, but it helps to equally distribute the transmit power between the in-phase (I) and quadrature (Q) components of the complex samples. Cleverly applied, it reduces the peak-to-average-power ratio (PAPR) of a signal. We calculate this value in dB of a signal  $s$  according to  $\text{PAPR}_{\text{dB}}(s) = 20 \log_{10} \frac{\max(|s|)}{\text{rms}(s)}$ . The PAPR is important, as energy efficient jamming requires to send tones with high average powers. Peaks in the baseband signal would require to reduce the signal's amplitudes to the dynamic range of the digital-to-analog converters (DACs), which automatically reduces the power in the transmitted signal.

*The more subcarriers we use, the lower the transmit power we can assign per subcarrier.*

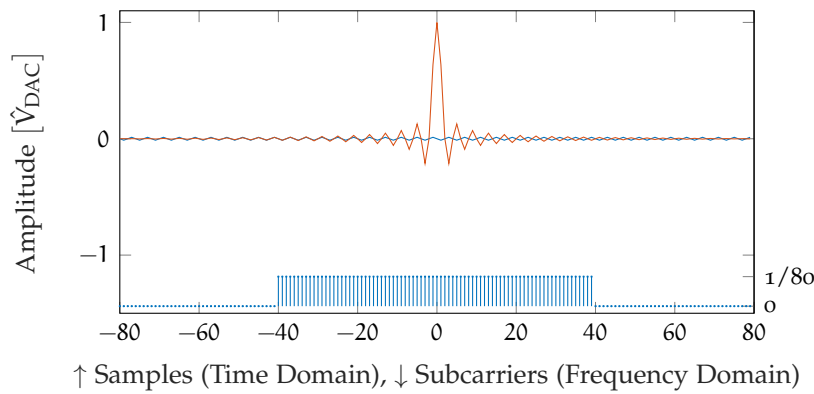
Figure 24 shows a 160-sample window of cyclically repeating time- and frequency-domain samples. An amplitude of one corresponds to the unknown peak voltage  $\hat{V}_{\text{DAC}}$  generated by the DACs. For Figure 24a we created a single tone by only setting one subcarrier leading to minimal PAPR of 0 dB and maximum power of 0 dBr (measured relatively to the output power of the DACs, i.e., before any amplification of the analog signals). Figure 24b shows the same for a signal with 80 active subcarriers. Due to the high PAPR of 19.03 dB the average power is only -19.03 dBr resulting in even less power per subcarrier. As a conclusion, we aim for jamming signals with few but high power subcarriers, similar to those used for jamming pilot tones.

#### 12.1.5 Signal amplification

After signal generation, the DACs create an analog waveform that needs amplification. The first step is a multiplication at the baseband multiplier (BBMULT) to increase the analog amplitude before upconversion in the mixers. We illustrate a block diagram of the Wi-Fi chip in Figure 2 on page 18. Then a chain of three adjustable amplifiers is used to amplify the radio frequency (RF) signal: (1) power amplifier



(a) Signal generated out of one subcarrier with normalized time-domain amplitudes leads to PAPR of 0 dB and average power of 0 dB.



(b) Signal generated out of 80 subcarriers with normalized time-domain amplitudes leads to PAPR of 19.03 dB and average power of -19.03 dB.

Figure 24: Peak-to-average-power ratio (PAPR) and average power analysis of single- and multi-tone signals. Each plot shows IQ time-domain samples at the top and the selected frequency-domain subcarriers at the bottom. (based on [74])

(PA), (2) power amplifier driver (PAD), and (3) programmable gain amplifier (PGA). To simplify gain selections, the firmware stores amplifier settings in a table addressable by an index. On the Nexus 5, the gain is mainly modified by setting the PGA value and slightly by setting BBMULT, while all other gains are set to maximum. Figure 25 shows gain settings for channel 7 (2.4 GHz band) and 106 (5 GHz band). The lower the index, the higher the gain. The exact output power of the smartphone is hard to measure without knowing the antenna radiation pattern and measuring received powers in an anechoic chamber.

*The baseband signal is amplified in the baseband multiplier, the RF signals traverse a power amplifier, a power amplifier driver and a programmable gain amplifier for amplification.*

#### 12.1.6 Power consumption

Especially on mobile devices, power consumption is of importance and influences how long a jammer can operate on battery power.

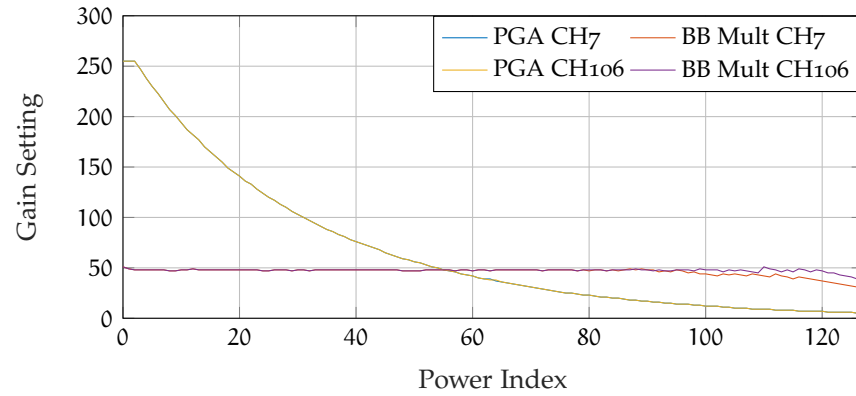


Figure 25: Programmable gain amplifier (PGA) and baseband multiplication (BBMULT) settings by index. (based on [74])

*We attach a Monsoon Power Monitor to measure the power consumption of our smartphone.*

*Without an active Wi-Fi chip, the Nexus 5 only consumes 16 mW in idle mode.*

*Power consumption of the Wi-Fi chip depends on the chosen Wi-Fi band and the bandwidth.*

*Continuous transmission of a test tone jacks up power consumption leading to a fast draining of the jammer's battery.*

Hence, we attached a Monsoon Power Monitor to the battery ports of a Nexus 5 smartphone to measure the instantaneous power consumption with 5 kSps. Using this setup, we first measured the phones idle power consumption. To this end, we turned off both the display and the LTE modem using a custom kernel as described in Section A.8. To minimize the activity of the Wi-Fi chip, we activate minimum power consumption (MPC) mode. MPC allows the ARM controller in the Wi-Fi chip to handle interrupts, while the radio, physical layer and D11 cores are on standby—neither receiving nor transmitting. In this mode, the phone consumes 16 mW. Turning MPC off enables the lower-layer cores in the Wi-Fi chip and activates the reception of Wi-Fi frames. As illustrated in Figure 26, this mode constantly consumes between 239 and 447 mW without transmitting anything. The power consumption increases with the channel bandwidth and by switching from the 2.4 to the 5 GHz Wi-Fi band. To collect the measurement results, we run experiments for 60 seconds, split the middle 40 seconds into 200 millisecond intervals over which we calculated the median after averaging to get an estimate of the power consumption. We use the median to reduce outliers due to non-deterministic tasks running on the CPU of the smartphone.

Afterwards, we measured the additional power required for continuously transmitting a 4 MHz test tone used for transmit signal strength indication (TSSI) measurements. We illustrate the results in Figure 27. Even using the smallest gains (largest power index), requires at least 556 mW in the 2.4 GHz band and 662 mW in the 5 GHz band. The difference is due to operating the analog front-end at higher frequencies which increases power consumption. To save power when operating the jammer, we focus our work on developing reactive jammers that only transmit short signals when required to destroy a frame. The power for operating the receiver is nevertheless continuously consumed. In the next section, we focus on the implementation of the three jammers.

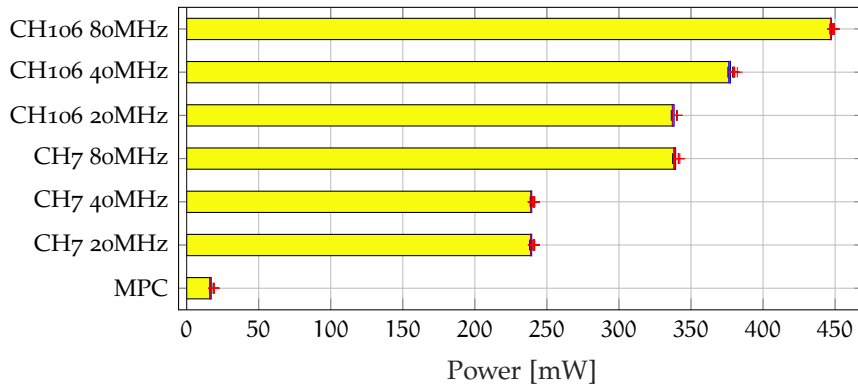


Figure 26: Power consumption for different channel specifications vs. power consumption with minimum power consumption (MPC) enabled. (based on [74])

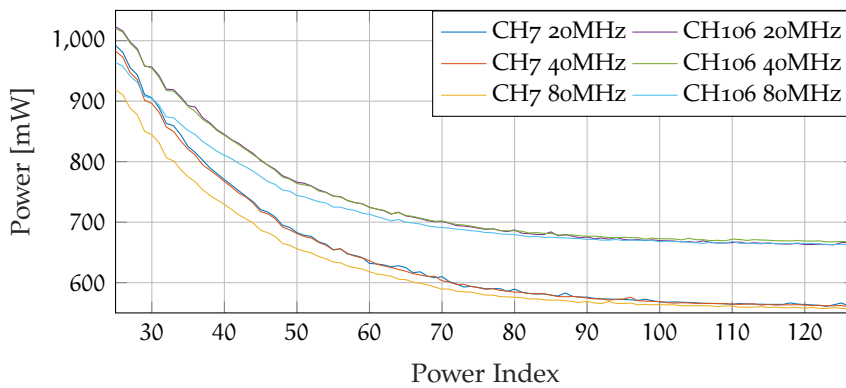


Figure 27: Power consumed by the smartphone for continuously transmitting the 4 MHz TSSI test tone at different power indices (minus the power consumption for operating the Wi-Fi chip with MPC disabled). (based on [74])

## 12.2 IMPLEMENTATION

To implement the jammers presented above, we, first, need to generate and load jamming signals into the sample-play buffer. To this end, we modify the firmware running on the ARM microcontroller and either generate hardcoded jamming signals or make them modifiable using settings passed through `ioctl`s. We describe this generic operation in Section 6.2.9. The simplest way to design a jamming signal on the smartphone itself, is our Nexmon Jamming app that offers an equalizer-like interface to set amplitudes and phases per subcarrier (see Section 12.2.1). After filling the sample-play buffer, we use the D11 core to trigger the transmission of the stored samples. To this end, the D11 core analyzes incoming Wi-Fi frames in real-time and tries to match jamming conditions (see Section 12.2.2).

We published the source code of the jamming firmware used in our experiments, as well as the source codes for both our jamming app

*Our implementation consists of code modifying the PSM in the D11 core, signal generation code for the ARM microcontroller as well as an optional app to control the firmware.*

*We publish the code of our tools to simplify its reuse and the reproducibility of our experiments.*

and its firmware (see Section A.4). Publishing the codes enhances reproducibility and eases reuse of our jammers in other projects. To avoid easy abuse, the jammer only targets frames with destination MAC address “NEXMON” and source MAC address “JAMMER”. Of course, this check can be disabled, but it requires an understanding of the D11 core’s source code. Below, we first present how our jamming app works and then go into detail of our jammer implementation in the D11 core.

### 12.2.1 Jamming app

*The jamming app combines a transmitter, a receiver and a jammer under one user interface.*

To get a better understanding of how a jammer operates, we created an Android app that allows to perform jamming experiments in a setup of at least three nodes. At least one acts as a transmitter that injects variable numbers of UDP streams with freely choosable modulation coding schemes. We offload the transmission of those frames into the Wi-Fi firmware, so that we can unburden the host from this task. To setup and control the transmission tasks, we use `ioctl`s. Another node acts as a receiver. It runs the Wi-Fi chip in monitor mode and counts frames that were injected by one of the transmitters. To differentiate which frames were jammed, we assume that in our setup only jammed frames are damaged. This implies the selection of a Wi-Fi channel with a low probability of collisions. As the frame check sequence (FCS) validation fails for damaged frames, we classify frames with an incorrect FCS as being jammed. In the user interface of the receiver, we create bar graphs for each UDP stream that illustrate how many frames were jammed and how many were correctly received.

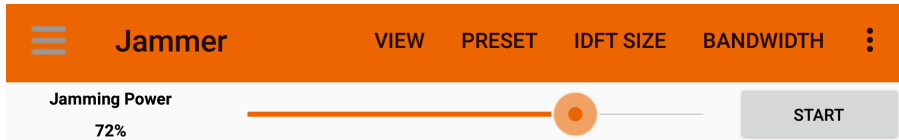
*The FCS helps to differentiate which frames are jammed.*

*The jammer UI offers sliders to set amplitudes and phases for the subcarriers of our jamming signal.*

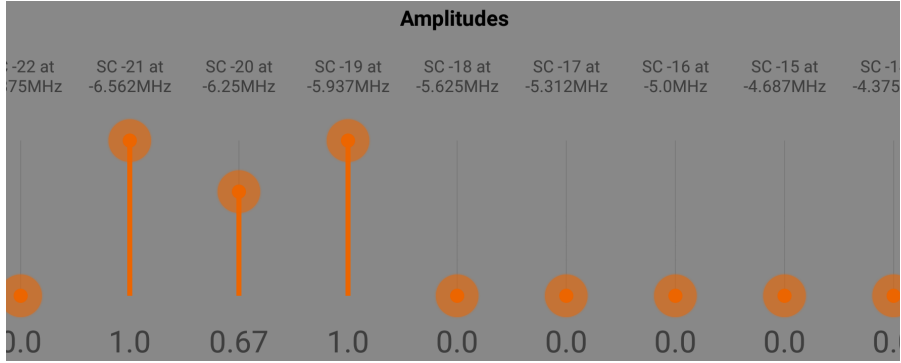
The most important component of our app is the jammer. It allows to configure the firmware and design jamming signals that can be transferred to the firmware using `ioctl`s. As described in Section 12.1.4, the jamming signals need to fit into the sample-play buffer and they need to be repeatable without introducing disturbances. Hence, we need to design our signals from tones whose periods fit completely into the buffer. The IDFT creates exactly those signals. For each of the subcarriers in the frequency domain, we can select an amplitude and a phase. The amplitude selection is illustrated in Figure 28b. To directly analyze how the generated waveform looks in the time-domain, we calculate the IDFT and plot the result showing the inphase and quadrature waveforms in Figure 28c. As the sliders do not give a good overview of the whole frequency band, we also plot the power distribution in Figure 28d. A click on the “START” button in the menu bar (see Figure 28a), packs the amplitude and phase values as well as other settings such as the transmission power into a byte array and calls the `nexutil` binary to send it—encapsulated in an `ioctl`—to the firmware. Using this interface, also other apps

*A clean and reusable interface based on `ioctl`s is used to configure the jamming firmware.*

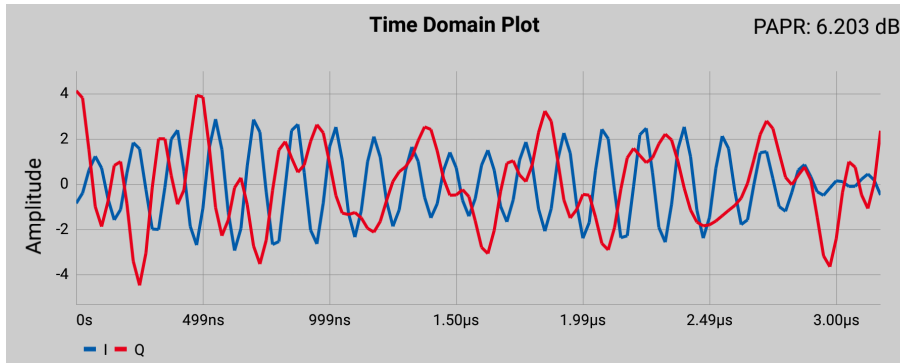




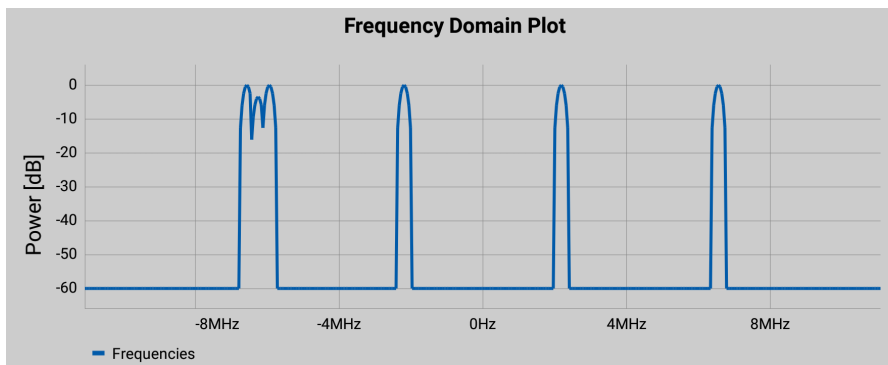
- (a) Menu bar of the jammer mode. Under VIEW, the user selects one or two of the UI-elements shown below.



- (b) In the *Amplitude* UI-element, the user sets the amplitudes per subcarrier of the signal used for jamming.



- (c) In the *Time Domain Plot* UI-element, we display the time-domain waveform resulting from the amplitude and phase settings. We also display the PAPR.



- (d) In the *Frequency Domain Plot* UI-element, we display the power distribution of the jamming signal.

Figure 28: The user interface (UI) of the jammer mode consists of a menu bar and four UI-elements to design or analyze the jamming signal (the *Phase* UI-element is not illustrated). (based on [78])

and console applications running on the smartphone can control the jammer as long as they have root permissions.

### 12.2.2 Implementation in the D11 core

*Modifying the ucode running on the D11 core is more complicated than changing the firmware of the ARM core, as we have to work with Assembler code.*

The heart of the implementation of any jammer running on the Wi-Fi chip is built into the ucode executed on the D11 core. While adding new ioctl handlers and calling functions to load values into the sample-play buffer on the ARM core is straightforward, the modification of the ucode is more advanced. As mentioned in Section 6.2.14, only a disassembler and an assembler exist to patch the ucode. To understand how the firmware works so that we can modify it, we first need to understand the meaning of values and registers. Fortunately, Gringoli and Nava published an open firmware for older Broadcom chips in [33]. The open firmware gives an idea on the operation of the ucode and parts of the code are similar to the disassembled version of the ucode running on the BCM4339 Wi-Fi chip. Based on this finding and the help of Francesco Gringoli, Michael Koch reverse engineered the ucode of the BCM4339 for his masterthesis [51]. Based on the reverse engineered ucode, we implemented our jammers.

### 12.2.3 Implementing the reactive jammer

*Reading our published source code helps following the implementation description.*

We first present an overview of the code and point at areas in the code we need to modify to implement the jammers. To follow the implementation description one should read the code referenced in Section A.4. The execution of the ucode always starts at address 0 in the code memory. One of the first instructions is always a jump to the `..._init` label that points to code for initializing the D11 core and the shared memory to a well defined state (the ellipsis `...` stands for a label number that may vary). Shortly below the jump instruction, we find code belonging to a state machine: `..._state_machine_...`. According to the comments in the OpenFWWF firmware [33], the state machine regularly checks conditions and jumps to labels of the corresponding condition handler implementations. One of these conditions is `RX_PLCP`. It is set as soon as a Wi-Fi frame is detected and the PLCP header is correctly received. Directly after the PLCP header start the bytes of a Wi-Fi frame beginning with the headers. As our reactive jammer should check whether header bytes match a jamming condition, the PLCP receive event handler is the optimal place to implement the condition matching functionality. After the first validation operations in the handler that should discard erroneous frames, we first implement a loop that waits until enough bytes are received to perform our condition matches. To this end, we check the `SPR_RXE_FRAMELEN` register against the required number of bytes. Then we start checking the conditions. If one matches, we directly call

*After successfully receiving the PLCP, we check our jamming condition and, if it matches, trigger a jamming signal transmission.*

the `transmit_jamming_signal` function that basically reimplements the `wlc_phy_runsamples_acphy` function (see Section 6.2.9) from the ARM code in the ucode. At the end, this function spin waits until the RX-to-TX sequencing is done to set another register. To avoid blocking the other ucode execution while waiting, we check the spinning condition every time we cycle through the state machine mentioned above. As soon as spinning is done, we set the required register. This fully implements the reactive jammer.

*In the ucode, we reimplement the ARM firmware's function to start transmissions.*

#### 12.2.4 Implementing the acknowledging jammer

Implementing the acknowledging jammer is more advanced. Directly activating the jamming signal in the PLCP handler would hinder us from successfully scheduling an acknowledgement transmission during frame reception. Hence, we have to first schedule the acknowledgement for transmission and then activate the jammer. To this end, we set the `SENDACK` shared memory variable in the PLCP handler instead of calling `transmit_jamming_signal`. Then, we need to make sure that an acknowledgement will be scheduled. To this end, we skip to the last part of the PLCP handler until we jump to a label we call `...rx_data_plus`. Here, the code spins until the MAC header is fully received (i.e., `SPR_RXE_FRAMELEN > 0x1C`) and then checks whether the frame contains our jammer's MAC address as receive address. Only if this is the case, it would jump to the `...send_response` label to schedule an acknowledgement. To also send an acknowledgement after jamming a frame with a different MAC address, we need to check whether the variable `SENDACK` was set before. If this is the case, we jump to the `...send_response` label ourselves. It prepares an acknowledgement with our frame's source address. Almost at the end of the `...send_response` function, we call the `transmit_jamming_signal` function to start jamming. After this function gets called, the D11 core waits until the `RX_COMPLETE` event occurs and jumps to the `...rx_complete` handler label. Its code finalizes the reception of a frame and checks the FCS to decide whether to send an acknowledgement or not. As we started jamming, the FCS will be incorrect. Hence, we have to force a jump to the label where execution continues if the `COND_RX_FCS_GOOD` condition matches. To actually transmit the acknowledgement frame, we have to enable FCS generation (0x4000) and enable the TX engine (0x1) by writing 0x4001 into the `SPR_TXE0_CTL` register. This fully implements the acknowledging jammer.

*For the acknowledging jammer, we first need to schedule an acknowledgement and then start jamming.*

*We need to force the transmission of an acknowledgement even though the jammer's MAC address differs from the destination address of the jammed frame.*

#### 12.2.5 Implementing the adaptive power-control jammer

The adaptive power-control jammer basically extends the acknowledging jammer by implementing the state machine illustrated in Fig-

*The adaptive power-control jammer only sends acknowledgements in the "JAM-ACK" state, otherwise it waits for acknowledgements of the target frame's receiver.*

*Using a timer, we check whether we received the expected acknowledgement to increase a counter checked while changing states.*

*We adjust the transmit power by changing the programmable-gain-amplifier value before starting to jam.*

ure 23. We implemented this state machine in the PLCP handler directly after checking whether the jamming condition matches. It counts how many frames were jammed in a state and compares those numbers against window lengths to decide whether to move to a new state. Instead of triggering the transmission of an acknowledgement after every jammed frame, we now check in the `..._rx_complete` handler whether we are in the "JAM-ACK" state or not. If we are, we send an acknowledgement, otherwise we set the `WAITACK` shared memory variable to mark that we are waiting for an acknowledgement transmitted by the target frame's receiver. To make sure that we detect the correct acknowledgement corresponding to our target frame, we check the time between finishing the reception of the target frame and the arrival of the next acknowledgement. Only if it falls within the bounds to expect acknowledgements, we count it as an indicator for an unsuccessfully jammed frame. To this end, we first save the value of the `SPR_TSF_WORD0` timer register into the `CLOCKACKTO` shared memory variable while we are in the `..._rx_complete` handler. We check this variable again against the timer register in the PLCP handler. If the difference falls in the expected bounds and we are in either the "LEARNING" or "CHECK" state, we increment the `ACKTRANSMITTED` shared memory variable corresponding to  $N_A$  in Figure 23. When changing states, we check this variable to decide on the next state and whether we should adjust the transmission power or not. To change it, we update the `PGA_NEXT` shared memory variable. It indicates the new gain value of the programmable gain amplifier (PGA). As illustrated in Figure 25, it is the main value changed by updating the power index. For simplicity, we only update this value while leaving the other amplifier values untouched. Whenever, we enter the `transmit_jamming_signal` function, we first write the `PGA_NEXT` value into the physical-layer table controlling the gains. This finalizes the implementation of our adaptive power-control jammer. In the following section, we evaluate the performance of our jammers.

### 12.3 EXPERIMENTAL EVALUATION

In this section, we describe the experimental setup and the experiments performed for the three different jammers directly followed by experiment evaluations and discussions. We intentionally chose a simple setup to ease reproducibility.

#### 12.3.1 Experimental setup

In Figure 29, we illustrate our experimental setup. It consists of one or two Nexus 5 smartphones used as frame transmitters and one Nexus 5 smartphone as frame receiver. The transmitters are used

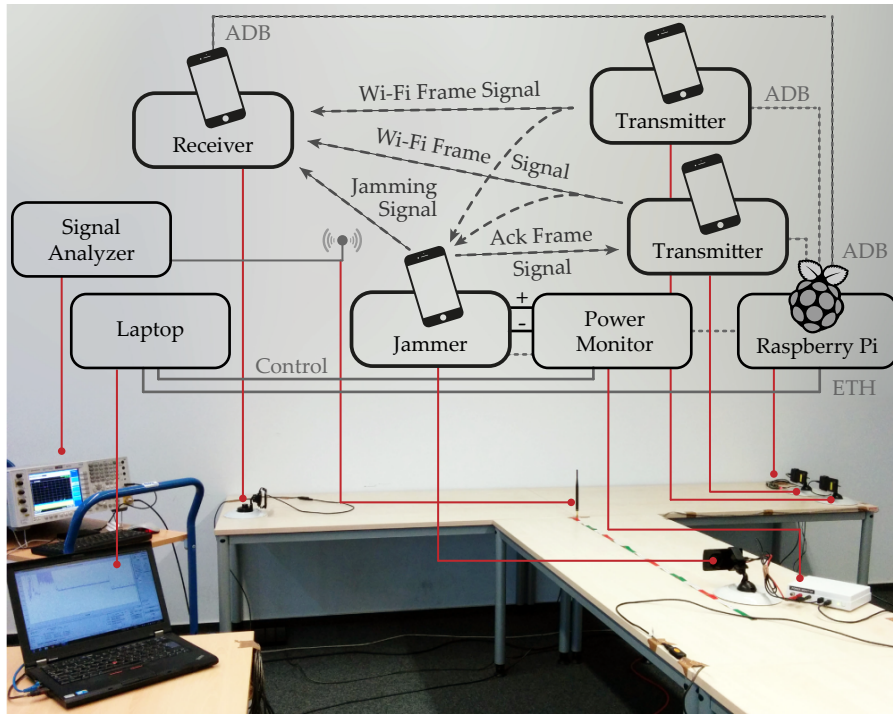


Figure 29: Experimental setup in our office building. The two transmitters, the receiver and the jammer are placed in the corners of an equilateral triangle. (based on [74])

to inject Wi-Fi frames with UDP payload at a fixed rate directly from the Wi-Fi firmware to avoid any jitter and additional queuing introduced by the operating system. Nevertheless, the injection complies to the 802.11 medium access control algorithms (i.e., CSMA). On the receiving node, we activate monitor mode and capture the received frames using tcpdump. We make sure that frames with incorrect frame check sequence (FCS) are also captured to be able to evaluate how many frames were corrupted either by the jammer (if the headers up to the UDP header are still correct) or by other effects that also destroy the frame headers. In general, we receive all frames, as long as the PLCP headers are correctly received.

As jammer, we also used a Nexus 5 smartphone with modified firmware for our jamming experiments. To evaluate the power consumption during jamming, we attached the Monsoon Power Monitor<sup>1</sup> to the battery ports of the jamming Nexus 5 and used a Laptop running Windows to capture the energy traces. All smartphones are controlled using the Android debugging bridge (ADB) over USB. We use a Raspberry Pi 3 as the controlling node to coordinate the experiments. After the measurement, USB is passed through the Power Monitor to the Raspberry Pi. For all experiments, the two transmitters, the receiver and the jammer are mounted using car mount holders attached to plates and placed in the corners of an equilateral

*We use the FCS to identify jammed frames at the receiver.*

*Using a power monitor attached to the jammer's battery port, we examine the power consumption while jamming.*

<sup>1</sup> Monsoon Power Monitor: <http://msoon.github.io/powermonitor/>

*Using a simple line-of-sight setup helps us to focus on the operations of the three jammer implementations.*

*A spectrum analyzer helps us to identify problems during experiments.*

*As the Nexus 5 does not contain a standardized antenna port, we focus on over-the-air experiments.*

*Successfully jamming requires a correct reception of the frame headers at the jammer.*

*In the first experiment, we deactivated the jammer, in the others we activated it with high transmission powers and run by run decreased those powers.*

triangle with side length of 2.8 meters. For increased side lengths, the received signal powers of both the data frame and the jamming signal equally decrease with distance (assuming line-of-sight behavior), hence, the jamming-to-signal ratios stay constant. Additionally, for massive jamming scenarios, we can assume high densities of smartphones usable for jamming, so that distances of only a few meters between nodes are likely.

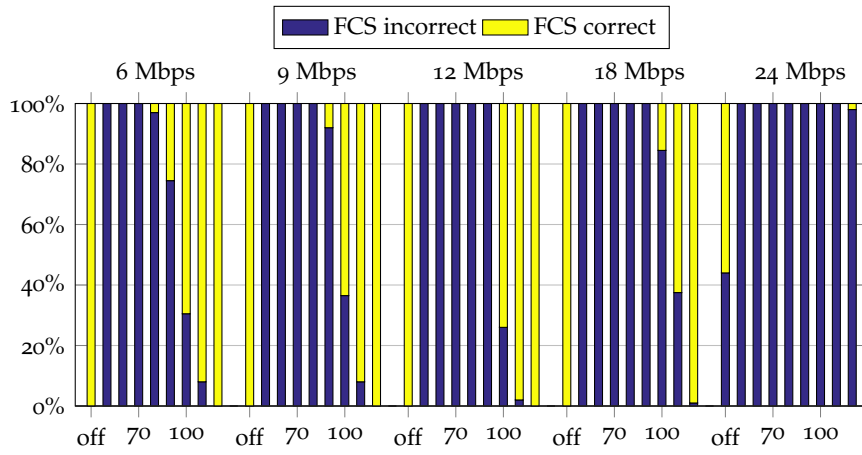
Even though not required to perform the experiments, we used a spectrum analyzer with 160 MHz real-time bandwidth and abilities to capture in the 2.4 and 5 GHz Wi-Fi bands to debug during development and to verify that the jammer works correctly. In the next subsection, we evaluate our reactive jammer.

### 12.3.2 Evaluating our reactive jammer

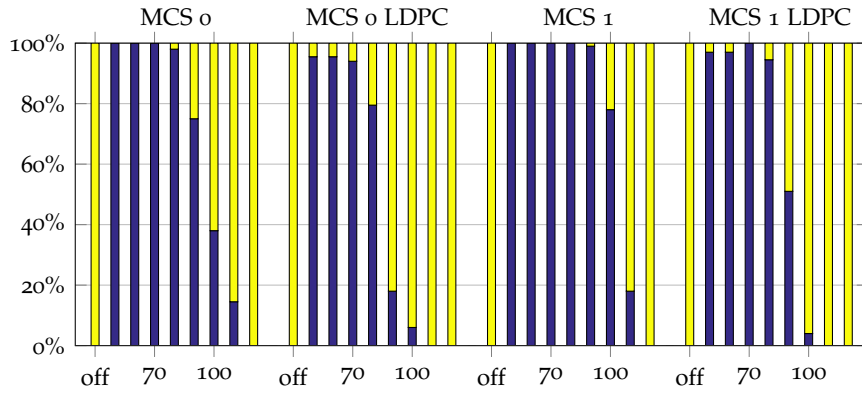
Our main intention in the evaluation of our reactive jammer is to verify the reliability and effectiveness of jamming with respect to the percentage of jammed frames at the receiver. As the Nexus 5 does not have a port for external antennas (that is matched to 50 Ohms used in measurement equipment), we focus our experiments solely on over-the-air jamming. This allows a realistic evaluation of the jamming performance in an office room and also incorporates antenna effects.

For successful jamming, two requirements need to be fulfilled. First, the jammer needs to correctly receive and decode the frames up to the part checked in the jamming condition. Secondly, the jamming signal needs to have sufficient power at the receiving node to interfere with the jammed frame. In a pre-evaluation of our jamming setup, we realized that direct communication between two Nexus 5 smartphones is prone to reception errors for high MCS settings, even in the small setup we used for our experiments.

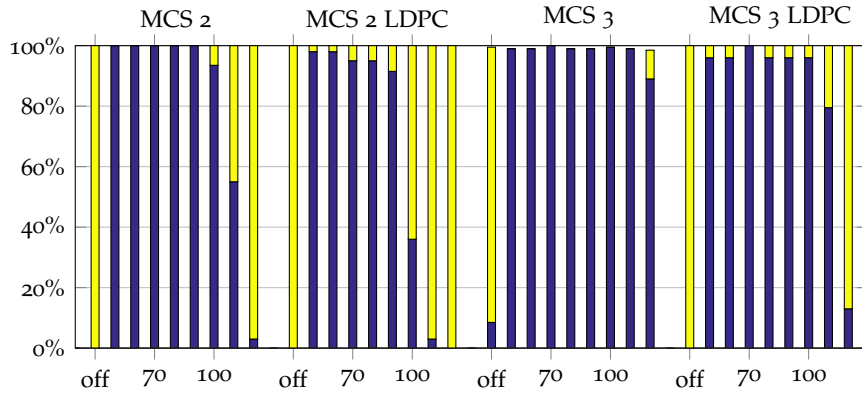
To evaluate the jamming performance, we sent frames without retransmission at different MCS settings from the transmitting smartphone next to the wall to the receiver at the other end of the long table (see Figure 29). In the first round, the jammer was deactivated to evaluate how many frames can be correctly received. In the other rounds, we started jamming at power index 50 (high power) and increased it in steps of 10 to 120 (low power). As jamming signals, we used pilot tones of 20 MHz bandwidth transmissions at subcarriers  $\{\pm 7, \pm 21\}$ . The results are illustrated in Figure 30. Each bar represents one round and shows the number of correctly received frames (FCS correct) and damaged frames (FCS incorrect). Starting from the legacy rate of 24 Mbps, the number of incorrectly received frames was already more than 40 percent. Increasing the MCS to 36 Mbps or MCS 4 for high-throughput rates, resulted only in erroneous frame receptions.



(a) Legacy 802.11a/g rates.



(b) First two 802.11n rates with and without LDPC.



(c) Next two 802.11n rates with and without LDPC.

Figure 30: Jamming frames on channel 116 in the 5 GHz band with 20 MHz bandwidth and different rates. The bars indicate correct frame receptions, when performing experiments with no jamming (off) and jamming at power index 50 (high power) to 120 (low power). The jamming-tone length is fixed to 128 us and the frame length to 1540 bytes. (based on [74])

*The use of power index 50 leads to successfully jamming most of the frames in our setup.*

Starting the jammer at power index 50 allows to corrupt all frames with deactivated low-density parity-check (LDPC) and even with the latter, close to 100 percent of the frames are jammed. Reducing the jamming power by setting higher power indices leads to the expected effect of a lower number of corrupted frames. 24 Mbps and MCS 3 transmissions are still very vulnerable to jamming even at these low transmission powers. Only LDPC can decrease the impact of jamming effects by providing better error-correction abilities. In the next section, we present how our jammer can be used in a practical friendly-jamming scenario.

### 12.3.3 Reactively jamming non-compliant 802.11ac transmissions

*Reactive jammers can be used to enforce standard-compliant behaviour as long as they can detect rogue communication.*

One application of friendly jamming is to counter the attack of malicious nodes that might setup rogue access points or communicate in a destructive way in a network environment. For example, a misbehaving pair of Wi-Fi devices might use non-compliant 80 MHz wide channels in the 2.4 GHz band that interfere with other communications in this shared band. Forcing the rogue nodes to switch to compliant transmissions can be achieved by jamming their communications so that their information exchange stops. Such a jammer, however, also needs the ability to receive Wi-Fi transmissions on illegal channel setups.

*Amplification needs to be adjusted to allow fair comparisons when jamming with different bandwidths.*

To evaluate this scenario, we deactivated the validation of channel specifications in the Wi-Fi firmware to be able to use 80 MHz wide channels in the 2.4 GHz band. Setting the carrier frequency to channel 9 (2452 MHz), covers the band from 2412 to 2492 MHz and, thus, almost the whole 2.4 GHz band. Then we evaluated, whether our jammer can receive and jam 20in80, 40in80 and 80in80 MHz transmissions using VHT MCS 0. 20in80 MHz means transmitting at the lowest of the four 20 MHz channels covered by the 80 MHz channel. 40in80 MHz means using the lower 40 MHz sideband, while the transceiver is still tuned to Wi-Fi channel 9. As jamming signals we used pilot tones at subcarriers  $\{-117, -103, -98, -75\}$  for 20in80 MHz,  $\{-117, -89, -75, -53, -39, -11\}$  for 40in80 MHz and  $\{\pm 103, \pm 75, \pm 39, \pm 11\}$  for 80in80 MHz transmissions. We adjusted BBMULT to achieve an equal power of all tones in all bands and set the tone transmission lengths to 128  $\mu$ s for 20in80 MHz, 64  $\mu$ s for 40in80 MHz and 32  $\mu$ s for 80in80 MHz transmissions.

*Our reactive jammer successfully destroys frames at bandwidths up to 80 MHz.*

We illustrate our results in Figure 31. Similar to Figure 30, the first bar indicates experiments without jamming. We observe, that transmissions are reliably received. During the experiments, the 2.4 GHz band in our office building was unused except for some beacon transmissions on channel 1. Starting the jammer at power index 50 (high power, second bar), all frames are corrupted at the receiver. Using lower transmission powers, the number of jamming successes reduce.



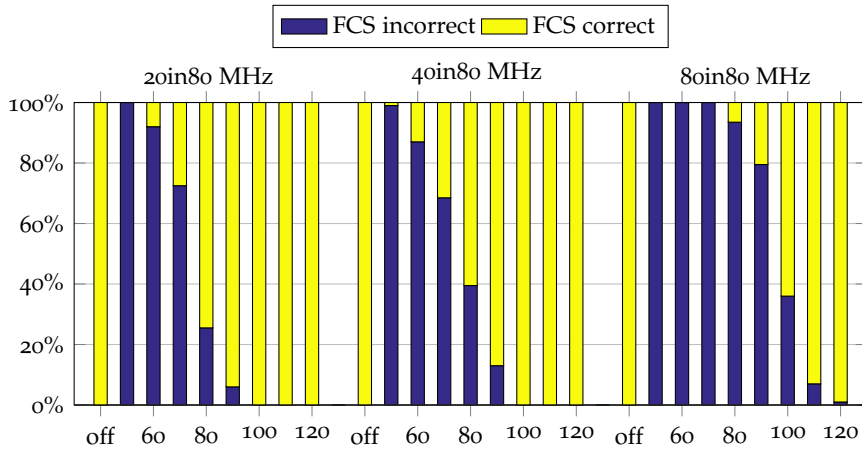


Figure 31: Jamming rogue 802.11ac transmissions on channel 9 in the 2.4 GHz band using 80 MHz bandwidth and sending 20, 40 and 80 MHz frames at VHT MCS 0. The frame length is fixed to 1540 bytes and the jamming-tone length to 128, 64 and 32  $\mu$ s for 20, 40 and 80 MHz bandwidth. The bars represent experiments with no jamming (off) and jamming at power index 50 (high power) to 120 (low power). (based on [74])

Most vulnerable are 80 MHz transmissions given our chosen jamming pattern.

#### 12.3.4 Multi-node jamming analysis

In the previous experiments, we focused on single link communications with only one active link and deactivated retransmissions. In this section, we extend our scenario by using two nodes sending frames in parallel on different UDP ports (3939 and 4040). Their MCS settings were fixed to 24 Mbps (OFDM) with five retransmissions for non-acknowledged frames. Starting from the third retransmission, we used a fallback rate of 1 Mbps (DSSS). All experiments were performed on channel 13 in the 2.4 GHz band. Additionally, we saturated the transmit queues in the Wi-Fi chips to send as many frames as possible.

At the receiving node, we again captured frames using tcpdump in monitor mode including frames with bad FCS. In Figure 32 we illustrate each experiment with one bar that represents the overall achieved UDP payload bit rate (including retransmitted frames). The bars are split to represent the bit rate dedicated to frames on port 3939 or 4040 additionally distinguishing between correct and incorrect frame check sums (FCSs). The first two bars show that transmissions with only one active node reach 18.2 Mbps. If two nodes are active at the same time, the throughput is split evenly and sums up to 18.6 Mbps. By activating our reactive jammer, the throughput of the jammed node vanishes, while the throughput of the second node

*For our experiments with multiple nodes we activated retransmissions and set conservative fallback rates.*

*Our reactive jammer selectively jams only the targeted frames, which can boost the throughputs of other nodes in a network.*

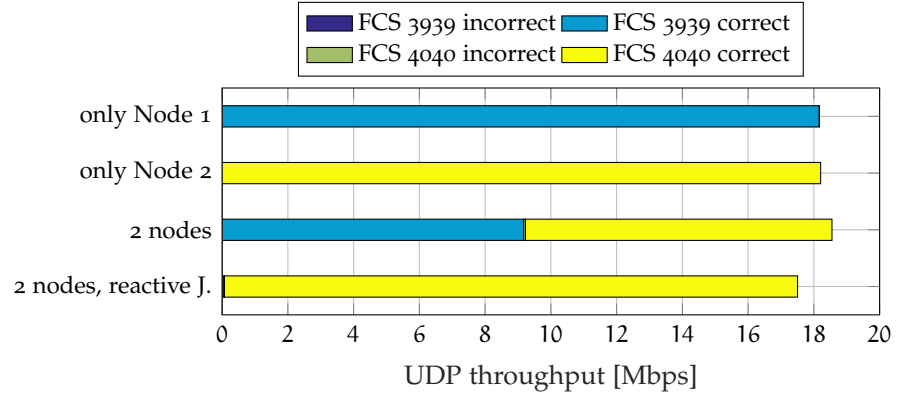


Figure 32: UDP throughputs in Mbps with either one or two active nodes using different UDP ports (Node 1: 3939, Node 2: 4040). Without jamming, the throughput splits evenly. Using the reactive jammer the throughput of Node 2 increases, while the throughput of Node 1 drops to a negligible rate of only incorrectly received frames. The jamming-tone length is fixed to 460.8 us to cover the complete remaining bytes of the 1540 bytes frame transmitted at 24 Mbps. (based on [74])

increases to 17.5 Mbps which is almost the rate a single transmitter achieves. In the next section, we change the setup to transmitting both streams with only one node.

#### 12.3.5 Flow-selective jamming

Instead of using reactive jammers to hinder a node from communicating completely, one may intent to specifically jam the communication of a certain service distinguishable by port numbers, while other services should still be available. As a usage scenario, we assume an industrial mesh network with legacy nodes that cannot be updated. The nodes offer multiple services of which one is vulnerable to remote code execution attacks. To protect the nodes while still being able to operate the non-vulnerable services, we employ our friendly reactive jammer. To evaluate this setup, we send two UDP streams on ports 3939 and 4040 from only one node. The results in Figure 33 show that the throughputs split evenly between the two streams, if no jammer is active. Using the reactive jammer, however, the whole throughput drops as the transmitter's MAC layer applies its backoff algorithm to all Wi-Fi frames—not differentiating between different upper layer streams.

To overcome this problem and allow communication on ports that are not jammed, we use the acknowledging jammer. Its operation is illustrated in Figure 34. Whenever a frame is received and the jamming condition matches, our jammer reactively jams it and also transmits an acknowledgement to the frames transmitter. Assuming correct re-

*By using multiple network services at the same time, the throughputs generally split between services.*

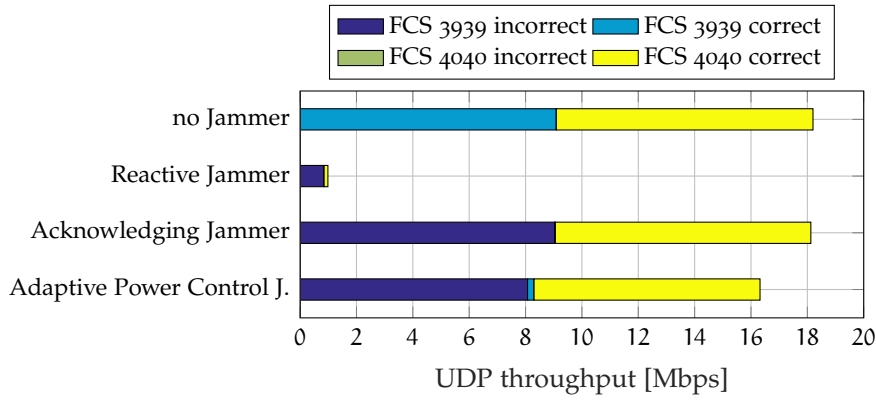


Figure 33: UDP throughputs with two streams sent by one node using UDP ports 3939 and 4040. Without jamming, the throughput splits evenly, using the reactive jammer kills the whole throughput, while the Acknowledging and adaptive power-control jammers still allow communication. The jamming-tone length is fixed to 460.8 us to cover the complete remaining bytes of the 1540 bytes frame transmitted at 24 Mbps. (based on [74])

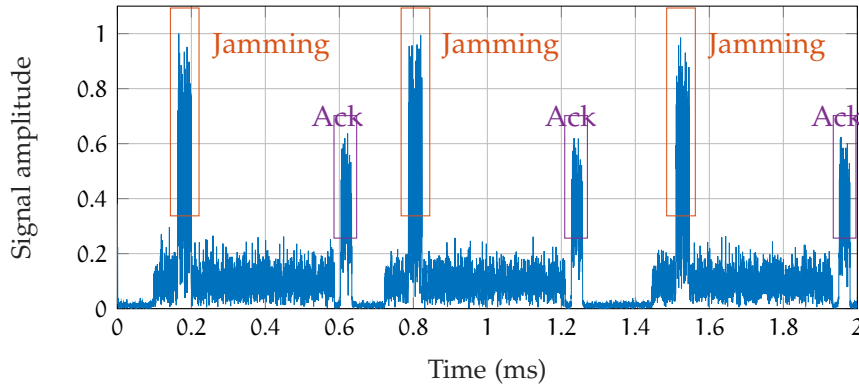


Figure 34: Operation of the acknowledging jammer. After jamming a frame, the jammer sends an acknowledgement to the transmitter indicating correct frame reception at the destination node. It takes roughly 20 us between receiving the UDP port number and sending the jamming signal.

ception of the frame, the transmitter can continue transmitting frames. The results are illustrated in Figure 33. Using the acknowledging jammer, frames to the jammed port 3939 are corrupted, while frames to 4040 are still correctly received and the throughput between jammed and non-jammed frames splits evenly. In the next section, we continue with a power consumption analysis for the presented jamming approaches.

*The acknowledging jammer only hinders the communication of targeted flows, while other data transmissions of the attacked node continue to flow.*

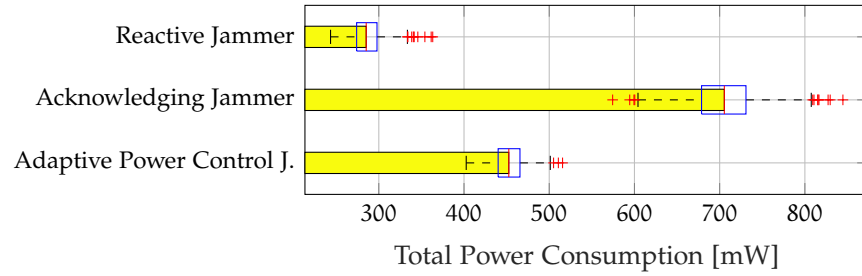


Figure 35: Total power consumption of the jammer, measured when jamming one transmitting node with two UDP streams (compare Figure 33). (based on [74])

### 12.3.6 Power consumption analysis

*The reactive jammer can benefit from backoffs between retransmissions and, thereby, consume less power than the acknowledging jammer that avoids retransmissions.*

To benefit from a smartphone's mobility, users rely on a low power consumption to maximize the time on battery power. Hence, we did a power-consumption analysis in parallel to the experiments described in Section 12.3.5. We illustrate the results for operating the jammer in our three implemented modes in Figure 35. As expected, the reactive jammer has the lowest power consumption of only 285 mW (resp. 30.7 hours runtime<sup>2</sup>), while 238 mW are allotted to operating the receiver with turned off MPC (see Figure 26). The low power consumption can be explained by the low number of frames that had to be jammed due to the increasing backoff at the transmitter for non-acknowledged frames. The acknowledging jammer consumes 705 mW (resp. 12.4 hours runtime), which is 2.5 times more power than operating the reactive jammer. The reasons for the higher consumption are twofold. First, the jammer needs to transmit an acknowledgement in addition to jamming. Secondly, the jammer needs to jam more frames as frames are transmitted at the highest rate without backoff delays.

*The adaptive power-control jammer can significantly reduce power consumption at the cost of some non-jammed frames while adjusting transmission powers.*

The best option to reduce the power consumption would be to reduce the transmission power. This, however, may lead to jamming misses, if the transmitter reduces its MCS settings to transmit more robust frames that are harder to jam with low jamming powers. To avoid this problem, we developed the adaptive power-control jammer, described in Section 12.1.3. It adjusts its jamming power according to the measured jamming success rate. In Figure 33, we observe that the adaptive power-control jammer also leads to equally split throughput for the two UDP streams. The total throughput is reduced to 16.3 Mbps and 2.8 percent of the targeted frames are not jammed. The correct frame receptions are due to the power adjustments. Whenever we reduce the power as a result of the "CHECK" state, a couple of frames may not be jammed until we increase the power again in the next iteration of the "CHECK" state. Nevertheless, the power con-

<sup>2</sup> Runtime calculation is based on standard LG BL-T9 batteries for the Nexus 5 with a typical energy of 8.74 Wh, respectively a capacity of 2.3 Ah.

sumption in our example setting reduces to 453 mW (resp. 19.3 hours runtime), which is 64 percent of the power consumed by the acknowledging jammer. As there is still room for improving the power adaptation algorithm, a user may optimize the jammer for either high jamming accuracy or low power consumption.

## 12.4 DISCUSSION

Our evaluation shows that smartphone-based Wi-Fi jammers are practical. Using our jamming app, one can get a hands-on experience on jammers for research and teaching. Additionally, we implemented advanced and new jamming methodologies such as the acknowledging jammer as well as the adaptive power-control jammer on a smartphone. The mobility requirements of smartphone users created the need for optimizations in power consumption to prolong operation on battery power.

During our experiments, we also came across a number of limitations of using the Nexus 5 as a reactive jammer. First of all, this smartphone has difficulties to correctly receive frames at high MCS indices, if they are transmitted by another Nexus 5 smartphone. A correct reception is, however, required to check jamming conditions based on the frame's payload. Without proper reception, jamming conditions are limited to the information provided in the PLCP header that is always transmitted at a robust rate. Additionally, the Nexus 5 has only one Wi-Fi antenna limiting its reception abilities to single-stream transmissions. Nevertheless, the Nexus 5 is sufficient for evaluating our three jammers in the field and it is possible to port our code to other more advanced Broadcom Wi-Fi chips as they are all based on the same architecture.

Overall, our work proves that sophisticated Wi-Fi jammers can be implemented in off-the-shelf smartphones. This allows to implement and massively distribute reactive jammers, as almost every one of us carries such a device around in daily life. Besides the friendly jamming applications proposed in this work to either block non-compliant devices or protect industrial networks containing otherwise vulnerable devices, omni-present jammers could also be used to perform wide-spread malicious attacks on our wireless infrastructure. As firmware is generally proprietary, users need to trust the firmware's developers that their hardware is not affected by malware that might transform somebody's phone into a remotely controlled jamming device. Only open firmwares and open specifications of radio devices would allow end users to verify by themselves that firmware running on their devices exhibits only benign behaviour. We present a thorough discussion of this aspect in Chapter 5. Knowing about the abilities of smartphone-based jammers at least allows to discuss possible solutions and countermeasures to avoid massive

*We successfully demonstrated that smartphone-based jammers are practical.*

*Nexus 5 smartphones are limited to SISO operation which can impede the reception of frames originating other SISO nodes.*

*The omni-presence of Wi-Fi enabled devices bears the risk of massive jamming attacks due to malicious firmware modifications that may spread from phone to phone in the form of a wireless worm.*

*Open firmwares are required to fix security issues after manufacturers drop support for their devices.*

attacks. These become more likely since the discovery of remote-code-injection vulnerabilities in Wi-Fi firmwares [7] that allow to remotely transform regular phones into jammers. Especially on old phones, these holes might never be fixed by manufacturers and due to the lack of open-source firmwares also not by the community.

## 12.5 RELATED WORK

*Various types of jammers are discussed in literature: (1) the constant jammer, (2) the deceptive jammer, (3) the random jammer, and (4) the reactive jammer.*

Jamming comprises a wide range of attacks and methods to distort wireless communication and prevent devices from either receiving or sending valid packets. Applied methods range from simple techniques (e.g., continuous transmission of interference) to more sophisticated approaches that exploit properties of higher layer network protocols [68]. Proposed jamming attacks differ in terms of disrupting impact, implementation complexity, energy consumption, stealthiness, and anti-jamming resistance [68, 96]. The prevalent types of jamming are (1) the constant jammer that transmits noise to corrupt frames or make the receivers sense a busy channel, (2) the deceptive jammer that is similar to the constant jammer but transmits arbitrary but valid protocol frames, (3) the random jammer that pauses a random time between transmissions to increase power efficiency, and (4) the reactive jammer [91] that targets only selected frames on the fly by detecting and corrupting them. All of these jamming strategies have been shown to have a significant impact but exhibit limitations in terms of efficiency, detectability, or resistance [68].

*Various authors present strategies to enhance efficiency and effectiveness by targeting modulation specific signal characteristics.*

Designing effective and efficient jammers is challenging. Constant and deceptive jamming have a high energy consumption that can be decreased by random jamming at cost of effectivity. The reactive jammer is efficient and effective but must handle strict real-time requirements: it must detect and distort a packet during transmission. A variety of research papers address efficiency aspects of jamming and show that smart jamming strategies can improve performance [70]. In [23], DeBruhl et al. apply game theory to develop energy-efficient jamming and anti-jamming strategies. [6] investigate the theoretical impacts of jamming IEEE 802.11 networks in [6]. Jamming techniques that disturb the pilot tones used for channel estimation and equalization have been shown to hinder receivers from decoding data packets [17, 36, 82]. In [91], Wilhelm et al. demonstrate the feasibility of reactive jamming on IEEE 802.15.4 networks in software-defined radio (SDR) based experiments. In [42], Vo-Huu, Vo-Huu, and Noubir demonstrate highly efficient reactive jamming and construct a jamming pattern across multiple OFDM subcarriers of IEEE 802.11a/g/n networks to be amplified by de-interleaving at the receiver. Therewith, they completely block a Wi-Fi communications with a jamming power of less than 1 percent of the communication power in an SDR-based testbed. However, SDRs are rather

*Jammers can exploit synchronization mechanisms such as pilot symbols and cheat the protective mechanisms of interleavers.*

expensive and cannot achieve performance results comparable to that of practical low-cost commodity hardware. This leads researchers to investigate jamming in common network devices.

In [88], Vanhoef and Piessens implemented a continuous and reactive jammer on top of an open source Atheros firmware. Although, having limited access to the firmware, they show that jammers can be implemented on commodity hardware. In [8, 9], Berger et al. investigated reactive jamming with off-the-shelf IEEE 802.11 access points. To that end, they directly modified the ucode of a SoftMAC Broadcom Wi-Fi chip. To this end, they extended the OpenFWWF firmware [33] published by Gringoli and Nava. Nexmon allows similar modifications, but aims at FullMAC chips deployed in mobile phones. There-with, it provides the ideal foundation to investigate mobile jammers in practical networks.

Besides for malicious purposes, jamming is also applied as a defense mechanism in the security context as “friendly jamming” or “jamming for good”. Recent work in this area utilizes jamming to either (a) block unauthorized communication or (b) secure confidential transmissions [8, 9]. To block unauthorized communication (a), reactive jamming can interfere with undesired frame transmissions and prevent them from being received at any receiver [12, 14, 31, 61, 83, 92, 95]. For example, in [31], Gollakota et al. protect implantable medical devices (IMDs) by jamming unauthorized commands. In [92], Wilhelm et al. propose a firewall for IEEE 802.15.4 networks that jams according to an arbitrary rule set. In [61], Martinovic, Pichota, and Schmitt develop message authentication mechanisms for wireless sensor networks (WSNs).

To secure confidential transmissions (b), jamming can prevent potential eavesdroppers from decoding a signal by causing artificial interference. This effectively allows only selected nodes to receive a confidential message. For example, Shen et al. control jamming signals with secret keys that only allow authenticated devices to recover transmitted signals but cause unpredictable interference to others in [83]. In [2], Anand, Lee, and Knightly induce interference into the null-space of a MIMO transmission and therewith jam all but the intended receiver. In [30], Gollakota and Katabi propose a physical-layer key-exchange based on random jamming patterns that distort parts of a transmission. In [50], Kim et al. use multiple jammers to form a physical secure area around access points. They adjust the jammers to jam everything outside to prevent information leakage. In [48] by Jorgensen et al., access points also mutually jam their transmissions to cause decoding errors at eavesdroppers. All these applications outline the valuable gain of jamming for protecting wireless networks.

Counter attacks on jamming are as vast as jamming itself. Metrics such as packet send ratio (PSR), packet delivery ratio (PDR), carrier

*Jammers have already been implemented on Atheros and SoftMAC Broadcom cards but only now on FullMAC chips available in smartphones.*

*“Friendly jamming” focuses on defensive applications to secure transmissions and block unauthorized communication.*

*Artificial noise is often used as a means to enhance the confidentiality of communication.*

*Jammers may even collaborate to cover wide areas.*

*Even though DSSS and FHSS should be hard to jam by narrow band interferers, defending against sophisticated jammers remains a hard challenge.*

sensing time, and received signal strength (RSS) can indicate whether jamming occurs or not [68, 96], but they cannot provide evidence alone. In [96], Xu et al. combine these metrics with consistency checks on the location and signal strength to reduce miss-detection rates. A common method to overcome narrow-band jamming is the application of direct sequence spread spectrum (DSSS) and frequency hopping spread spectrum (FHSS) as presented in [65] by Mpitiopoulos et al. Lin and Noubir propose enhanced coding schemes to prevent jamming and establish jamming resilient communications in [58]. In [97, 98], Yan et al. utilize techniques from interference cancellation and signal processing to maintain MIMO-OFDM communications under reactive jamming. To this end, they align the legitimate signal orthogonal to the jamming signal. Still, counter measures are costly, and launching and defending jamming attacks remains an arms-race [68].

## 12.6 FUTURE WORK

*Future work could enhance our jammers to support different jamming waveforms or simplify the adaptive power-control jammer's state machine by duplex capable hardware.*

*The use of other Nexmon-based technology such as wide-band transmission or CSI extraction capabilities could lead to further improvements and new applications.*

While this work focuses on new jammer enhancements and their implementation and evaluation on off-the-shelf smartphones, we by far do not cover all the possibilities smartphone-based jammers could offer. Incremental improvements could be a better state machine to further reduce power consumption and enhance stability. Duplex capable Wi-Fi chips could even detect the presence of an acknowledgement while transmitting a fake acknowledgement. This would make “LEARNING” and “CHECK” states superfluous as they could be performed in parallel to the “JAM-ACK” state. Additionally, the effectiveness of other jamming waveforms should be evaluated, for example, to support interleaving jamming [42] on smartphones. To this end, different jamming patterns need to be prepared to cope with every possible scrambler seed. This demands for larger sample storage memories such as the Template RAM from which we can also transmit signals as presented in Chapter 9. The unleashed capability to cover 80 MHz bandwidth in the 2.4 GHz band also allows to target wide-band frequency hopping schemes such as Bluetooth to create something like a cross-technology jammer. Additionally, physical-layer information extracted from received frames may help to enhance our jammer. As an example, the channel-state information read by our Nexmon CSI extractor presented in Chapter 8 helps to focus jamming power on subcarriers that best traverse the channel between jammer and intended receiver. For this application, we would need to extract the CSI of a frame originating the intended receiver. Alternatively, CSI fingerprints could enhance reaction times of reactive jammers as jamming conditions could be evaluated directly after receiving the preamble containing the LTF used for CSI extraction. The examples above show, that a lot of research with jammers on smartphone is still possible.



## 12.7 CONCLUSION

In this chapter, we proved that one can easily transform off-the-shelf smartphones into efficient, mobile jamming devices. We demonstrated that small modifications to the firmware running in the Wi-Fi chip allow us to quickly react on frame receptions to transmit jamming signals in time to destroy the frame during its transmission. We are, thereby, not limited to predefined waveforms, but can design our own signals as IQ samples that are directly injected into the baseband. This opens the possibility to use such cheap devices in place of more complex software-defined radio platforms. This flexibility clearly poses a serious threat. If a malicious attacker manages to inject a modified firmware into a large number of devices, he could launch tremendous, distributed attacks against networks in the 2.4 and 5 GHz bands. We, instead, used the flexibility for friendly jamming applications and presented innovative jamming techniques that involve the transmission of a forged, matching acknowledgement to cheat the transmitter into believing that no transmission was actually jammed. Together with a proof-of-concept prototype that automatically determines the optimal transmission power, we presented the first 802.11ac compliant and energy efficient personal jamming platform.

*With our work, we demonstrated that reactive jammers that send arbitrary waveforms no longer rely on SDRs.*

*With the acknowledging jammer we introduced a novel approach to allow flow-based jamming without throttling all flows of a targeted transmitter.*

## 12.8 MY CONTRIBUTION AND ACKNOWLEDGEMENTS

I came up with the idea to port Francesco Gringoli's jamming firmware [8, 9] from legacy Wi-Fi chips to smartphones to support mobile reactive jamming. I thank Michael Koch for working on this topic and for reverse engineering the BCM4339 ucode in his masterthesis [51]. In parallel, I understood how to transmit signals from the sample-play buffer and had the idea to use this capability to generate arbitrary jamming signals. To work on this idea, I started a collaboration with Francesco Gringoli. I thank him for this intensive collaboration and especially for writing the D11 code that implements the three jammers. During the implementation, I came up with the ideas of the two advanced jamming types. For their implementation, I also relied on Francesco Gringoli's profound knowledge on Broadcom's ucode. Then I had the idea to extend the work and make it accessible by providing a jamming app. I thank Efstathios Deligeorgopoulos for implementing this app. Additionally, I thank Daniel Steinmetzer for collecting related work and writing up the related work section.

*Only our collaboration let us achieve the greater goal.*



As a proof-of-concept application that uses both SDR-like transmissions and CSI extraction capabilities, we chose to implement a new physical-layer-based covert channel. By means of this covert channel, it is possible to stealthily embed additional information into Wi-Fi frames. Ideally, this should not impact the reception of such frames by normal receivers. Yet, it allows to covertly exchange information between two devices Alice and Bob that can observe each others radio communications. Similar to most physical-layer covert channels, this channel can be detected and decoded by an eavesdropper (Eve), in case she knows the implementation details of the channel and has access to similar SDR-like functionality or advanced signal analysis capabilities on the physical layer. In contrast, this channel will be indiscernible for unmodified off-the-shelf devices, which do not allow for user access to physical-layer parameters. To demonstrate the advanced Wi-Fi capabilities of our solution, we introduce Shadow Wi-Fi, a physical-layer-based covert channel over Wi-Fi. The covert channel itself works by pre-filtering outgoing Wi-Fi frames and encoding secret information into the filter. We can observe the filters' effects at a receiver by evaluating the per-frame channel state information. During the development, we faced the following challenges: (1) we needed a way for prefiltering outgoing frames in the Wi-Fi chip, (2) we had to extract per-frame channel state information, and (3) we had to trigger transmissions from Template RAM from within the ucode.

In this chapter, we first describe the design of our covert channel in Section 13.1, followed by its implementation in Section 13.2. We, then, evaluate our implementation using two Nexus 5 smartphones in Section 13.3, followed by a discussion in Section 13.4. Then, we present related work in Section 13.5 and future work in Section 13.6 and finally conclude in Section 13.7.

### 13.1 COVERT CHANNEL DESIGN

Our covert channel relies on the ability of every Wi-Fi receiver to cope with the fading effects of the wireless channel to reconstruct the originally transmitted data symbols plus noise introduced during the transmission. To this end, every OFDM-based Wi-Fi receiver extracts channel state information (CSI) that describes—for each subcarrier—how the wireless channel changed amplitudes and phases. We describe this receive operation in more detail in Section 4.3.2. By introducing an additional fading-like effect using a transmit filter, we can

*Using the Nexmon SDR, we transmit modified Wi-Fi frames with embedded covert information and use the Nexmon CSI Extractor to read this information at a receiver.*

*Triggering transmissions from Template RAM in the ucode was more challenging than expected.*

*Each Wi-Fi receiver has to cope with fading effects, that we may extend to embed secret information into a Wi-Fi frame.*

secretly embed additional information into each Wi-Fi frame without destroying the ability of a regular Wi-Fi receiver to correctly receive the frame. In what follows, we present in detail how our covert channel works. To get an overview of the system, we illustrate our covert channel as a block diagram in Figure 36.

*The received symbol on each subcarrier is just the transmitted symbol multiplied by the channel coefficient—one entry in the CSI vector.*

*We can multiply each outgoing symbol with a transmit filter coefficient to change the CSI measurement at a receiver.*

*Filter coefficients modifying only the phase of an outgoing symbol lead to signal changes that are harder to detect than changes in the amplitude.*

*We filter in the time domain to generate inter-symbol interference similar to wireless channels.*

Looking at our covert channel more formally, the effect of the wireless channel  $H_{sc}$  on a subcarrier  $sc$  can be linearly applied to the transmitted QAM-symbol  $X_{sc}[k]$  of the  $k$ -th OFDM-symbol, resulting in a received symbol  $Y_{sc}[k] = H_{sc} \cdot X_{sc}[k]$ .  $H_{sc}$  is assumed to be constant during the transmission of one Wi-Fi frame. Hence, receivers extract it once per frame from the long-training field (LTF) resulting in the CSI measurement. Instead of transmitting the  $X_{sc}[k]$  symbols directly, we can apply a filter  $F_{sc}$  to all transmitted symbols first, including those of the long-training field resulting in received symbols  $Y_{sc}[k] = H_{sc} \cdot F_{sc} \cdot X_{sc}[k]$ . As each Wi-Fi receiver estimates the CSI based on the channel effects applied to the LTF, it simply considers the transmit filter as part of the wireless channel  $H'_{sc} = H_{sc} \cdot F_{sc}$  and automatically cancels its effect when decoding data signals. To simplify the equations, we combine variables depending on the subcarrier  $sc$  into column vectors, for example,  $\vec{H} = (H_0 \cdots H_{\max(sc)})^T$ , leading to  $\vec{Y}[k] = \text{diag}(\vec{H}) \cdot \text{diag}(\vec{F}) \cdot \vec{X}$ .

To build a covert channel that does not disturb the regular Wi-Fi communication, we use differentiable filter vectors  $\vec{F}_x$  as secret symbols. Hence, we can embed one secret symbol per frame. In the filter vectors, we can change amplitudes and phases of all transmitted symbols on all subcarriers. While changes in the amplitude are easily observable using spectrum analyzers, the detection of phase changes requires more thorough signal processing steps which impedes adversaries from easily detecting a covert transmission. Hence, we focus on modifying only the phases of particular subcarriers in transmitted frames. To mimic the behaviour of passing each Wi-Fi frame through an additional wireless channel when we apply the filter to embed covert symbols, we first convert our filter vector into the time-domain to create the filter's impulse response  $f(n)$  and then apply it by convolution with the Wi-Fi frame's waveform  $x(n) * f(n)$ . This increases the risk for inter-symbol interference (ISI), but it makes sure that this additional signal processing effect does not end on OFDM-symbol boundaries as it would be the case for filtering in the frequency domain. This hinders an adversary from gaining another feature he may look for when searching for irregularities that may imply the existence of a covert channel. In the next section, we focus on the implementation of our covert channel in Wi-Fi chips and cope with practical problems.

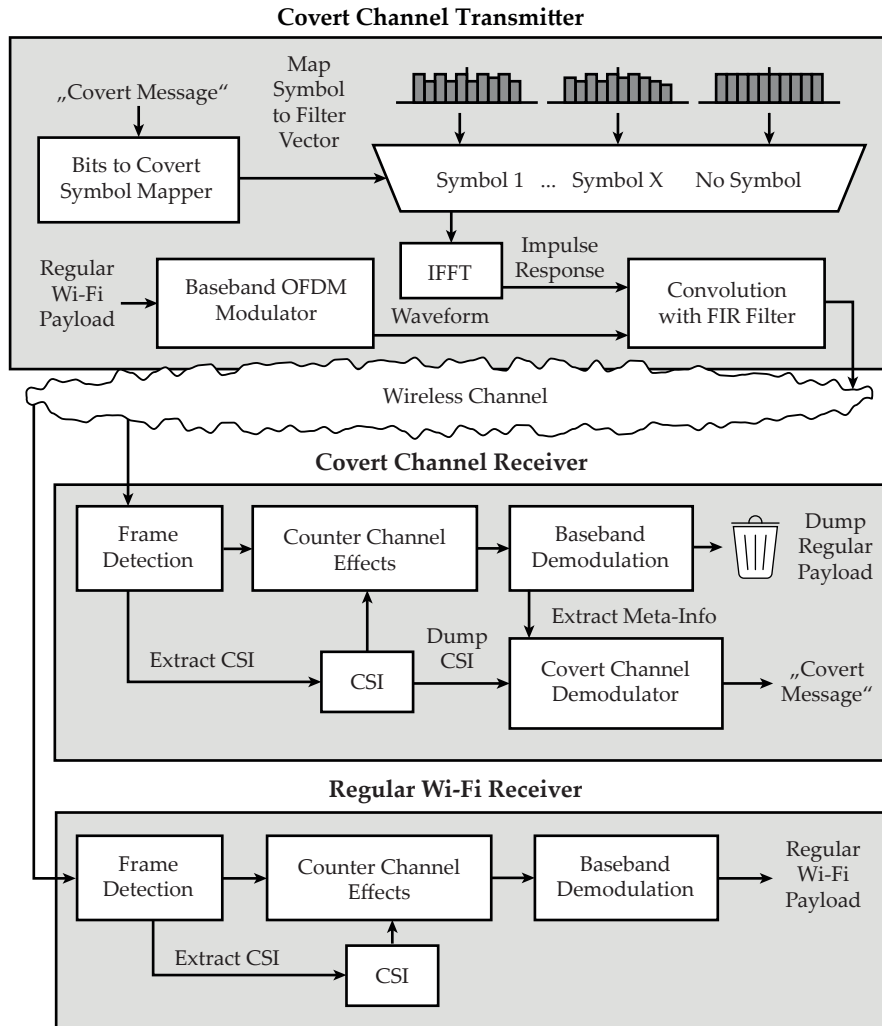


Figure 36: In our covert channel, the transmitter first maps message bits to symbols that select filters that we apply to outgoing frames by convolution in the time domain. A regular receiver estimates the transmit filter as part of the wireless channel so that its effect will be canceled. While a covert channel receiver may extract the hidden message from the measured channel state information. (based on [75])

## 13.2 IMPLEMENTATION

Currently, we are not aware of a finite impulse response (FIR) filter in the transmit chain, that we could easily adjust on a per-frame basis to filter all outgoing frames that are created in the Wi-Fi baseband modulators. A possible solution to circumvent this very efficient implementation option, is implementing the covert channel transmitter by handcrafting the complete Wi-Fi raw signals, filtering them and transmitting them with the Nexmon SDR presented in Chapter 9. As this approach is very time consuming and does not support high

*As we have no knowledge of easily accessible FIR filters in the Wi-Fi chip, we filter handcrafted Wi-Fi frames manually.*

frame rates, we decided for an optimized implementation that also works in real-time systems.

We assume that our covert channel transmitter is communicating with another node, for example, an access point. Instead of embedding the covert channel in the outgoing data frames—that are constantly changing—we embed the covert channel into the outgoing acknowledgements. As acknowledgements are always addressed to the same communication partner (e.g., the access point), every frame contains exactly the same payload. Hence, we can generate prefiltered acknowledgement frame signals and store them in the Wi-Fi chips Template RAM. The higher the number of differently prefiltered acknowledgements we generate, the more covert symbols we can transmit. To additionally reduce the time for extracting CSI information, we can optimize our CSI extractor to only copy the CSI values of subcarriers used by our covert channel implementation.

*As handcrafting every data frame is computationally expensive, we exchange covert symbols using prefiltered acknowledgement frames whose content is static.*

### 13.2.1 Generating and sending acknowledgements with covert information

For Wi-Fi frame generation, we use MATLAB's WLAN System Toolbox. It allows generating frames with arbitrary payload at all known modulation coding schemes (MCSs) and bandwidths. We use this toolbox to generate our acknowledgement frames and then apply various filters in the time domain to embed covert channel symbols. Then, we load the generated symbols into the Wi-Fi chip's Template RAM and transmit them as a response to frames received by the communication partner. To answer with raw-signal acknowledgements from the D11 core, we have to intercept each frame reception after receiving the PLCP header and spin wait until we complete the reception of the MAC addresses. This implementation is similar to the one of our jammer presented in Section 12.2.3. If we received a frame for us sent by our communication partner, we skip scheduling the transmission of a regular acknowledgement and set a variable that indicates to transmit a raw-signal acknowledgement. Then we wait until the frame is completely received, confirm that the frame check sequence (FCS) is correct and trigger a raw-signal transmission from Template RAM. To this end, we lookup start and end pointers in shared memory that indicate where the raw samples are stored in Template RAM and write them into the corresponding registers. Then we trigger the playback of these samples and perform the RX-to-TX sequencing to activate the transmitter. Then we spin wait until the end of the transmission and reset the clear channel assessment to stop the transmission and return the D11 core and the physical layer to a state where they can receive the next frame.

*We use MATLAB to generate raw samples and acknowledgement frames and then load them into Template RAM for transmission.*

*To trigger the transmission from Template RAM, we extended the D11 ucode to trigger a raw transmission instead of the regular acknowledgement.*

At the covert channel receiver, we dump the CSI information of all acknowledgement frames. Then, we analyze the CSI dumps to extract the covert symbols. In a realistic setup the receiver has to

synchronize on the transmitted symbols to identify where a covert channel transmission starts and where it ends. We implement a simple communication protocol for this purpose. We assume that our secret information can be split into messages that contain exactly one byte. To indicate the start and end of a covert byte, we use special message start and stop symbols. On the transmitter side, we store secret messages as sequences of start and stop pointers in the shared memory that point to prefiltered acknowledgements containing different covert symbols. For each acknowledged frame that contains a new sequence number, the transmitter iterates over the shared memory entries to transmit the stored messages.

Last but not least, we define how covert symbols are modulated. We basically have three options to modulate symbols on each subcarrier: (1) changing the phase, (2) changing the amplitude, or (3) changing both of them. By influencing the amplitude, we change the average power on a limited number of subcarriers. Even though a regular multi-path channel may have a similar effect, the spectral shape of the transmitted frames may deviate from standard Wi-Fi frames, so that the existence of the covert channel is easily detected by looking at the power spectral density or the amplitude of the CSI on a line-of-sight channel. By modulating the phase, instead, we can achieve a better covertness of our channel, as its detection requires a look at phase changes in the CSI. Simply investigating transmitted signals with a spectrum analyzer is not sufficient anymore to detect the existence of this channel. Hence, we focus only on phase changes.

*For receiving messages that span over multiple frames, we use a simple protocol indicating the start and end of each message.*

*We can modulate cover symbols by changing amplitudes, phases or both.*

*Modulating only the phase keeps the power spectral density of regular Wi-Fi frames and is, hence, harder to detect.*

### 13.2.2 Choosing covert symbols

We generate 16 different covert symbols ( $0000_2, 0001_2 \dots 1111_2$ ). Each of them is a filter vector in the frequency domain that changes the phases of the four subcarriers  $8 + s$ ,  $11 + s$ ,  $54 - s$ , and  $58 - s$  by  $170^\circ$ ,  $190^\circ$ ,  $170^\circ$ , and  $190^\circ$ , where  $s \in [0, \dots, 15]$  is the symbol index. We transform these vectors into the time domain by using an IFFT to generate the filters' impulse responses that we can apply to Wi-Fi frame signals by convolution. Our frames are 802.11g modulated at all eight modulation coding schemes (MCSs), resulting in the bit rates from 6 Mbps to 54 Mbps. We generate those frames in MATLAB and filter each of them with our 16 covert symbols, which results in 128 acknowledgement frames with embedded covert symbols that we use for the experiments described below. To ease our analysis, we also write a covert symbol identifier into the last two bytes of the frames' MAC addresses. Obviously, for real covert channel operation, this must be avoided.

*We generate 16 covert symbols that changes phases on selected subcarriers.*

*We end up with 128 acknowledgement frames at different 802.11g rates.*

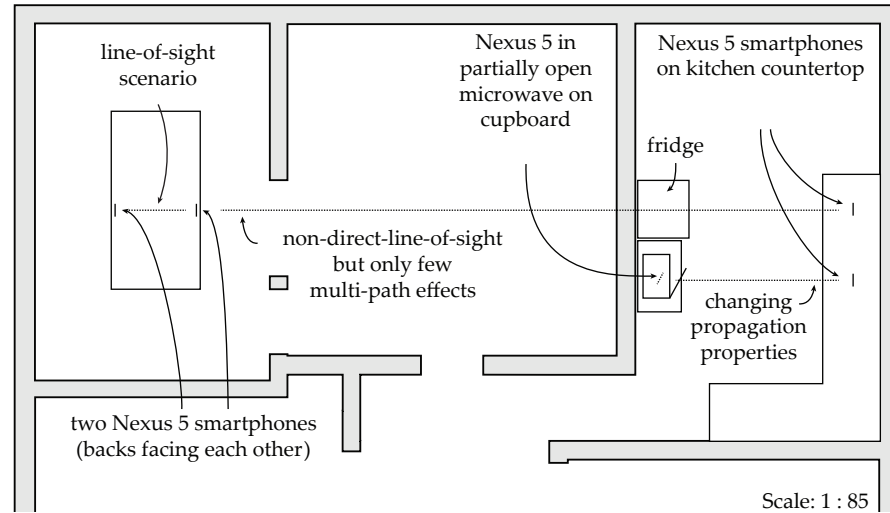


Figure 37: Experimental setup in an apartment in a rural environment with no other Wi-Fi traffic on channel 122 and 120. Placing the transmitter in the partially open microwave increases multi-path effects. All smartphones are installed on car mount holders to enhance the antenna radiation characteristics. (based on [75])

### 13.3 EXPERIMENTAL EVALUATION

*We perform our experiments in an apartment in a rural area that has a low amount of Wi-Fi traffic.*

In our experimental evaluation, we study the performance of our covert channel implementation and directly discuss the experimental results. As environment we choose an apartment in a rural area with only low amount of Wi-Fi traffic so that our experiments are not affected by high levels of interference. Even by communicating between rooms and shielding the line-of-sight paths with a refrigerator, we only observed low multi-path effects in our CSI measurements. For experiments that do not focus on multi-path propagation, we simply stayed in one room. For influencing the propagation characteristics of the wireless channel in a repeatable fashion, we placed one device into a microwave oven and changed the opening angle of the microwave door. We document our different communication setups in Figure 37.

*We evaluate three different aspects.*

To evaluate the covert channel performance, we look at three different aspects. First, we evaluate how embedded covert symbols influence the frame reception at normal Wi-Fi receivers. Second, we measure covert symbol detection ratios at the covert channel receiver. Third, we perform a realistic message exchange using our covert channel and demonstrate that it is practical.

#### 13.3.1 Covert channel experiment in line-of-sight setup

In our first experimental setup, we place two Nexus 5 smartphones in a distance of 1 m on a table top with the backs facing each other (see



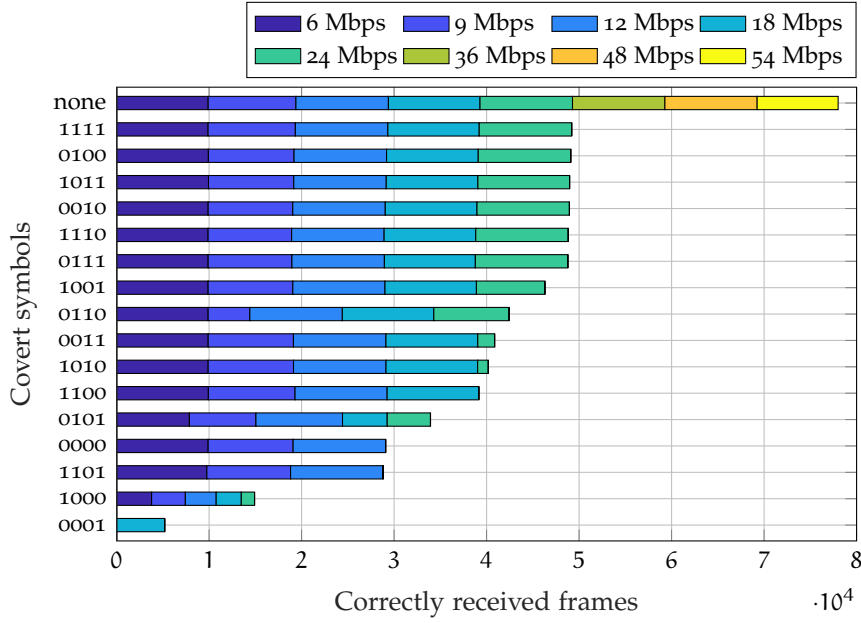


Figure 38: Frame reception rates at off-the-shelf Wi-Fi receiver with different covert channel symbols that each flip the phases on four subcarriers. (based on [75])

Figure 37). We use the phone on the right as covert channel transmitter. For each sub-experiment, we first load the raw samples of one of the generated acknowledgement frames into the Template RAM and then start a timer that triggers the transmission of 10 000 raw frames—one per millisecond. We use the second Nexus 5 smartphone on the left as both off-the-shelf Wi-Fi receiver and covert channel extractor to simplify the setup by saving a third node. To this end, the phone always dumps the received acknowledgement frame as well as the channel state information into a pcap file. After repeating the experiment for all 128 pre-generated frames, we analyze the pcap contents.

*We first transmit all generated acknowledgements 10 000 times and dump them including their CSI for further analysis.*

### 13.3.2 Evaluating the influence on normal Wi-Fi receivers

First, we analyze the effect of the embedded covert symbols to the regular Wi-Fi receiver. We count frames per MAC address. As this address contains an identifier of the embedded covert symbol, we directly know which frames are modified and how they were pre-filtered. In Figure 38, we present the results. Each row illustrates the number of received frames during the eight experiments—one for each MCS. The first row is the result without covert symbols. Here, the maximum of 10 000 frames per sub-experiment was almost reached. The next six rows belong to covert symbols that have no measurable influence on the reception performance for MCSs of up to 24 Mbps. They are, hence, a good choice for embedding covert symbols without raising suspicions at covert channel detectors that

*Covert symbols that have a low effect on the reception performance of a regular Wi-Fi node help us to stay covert.*

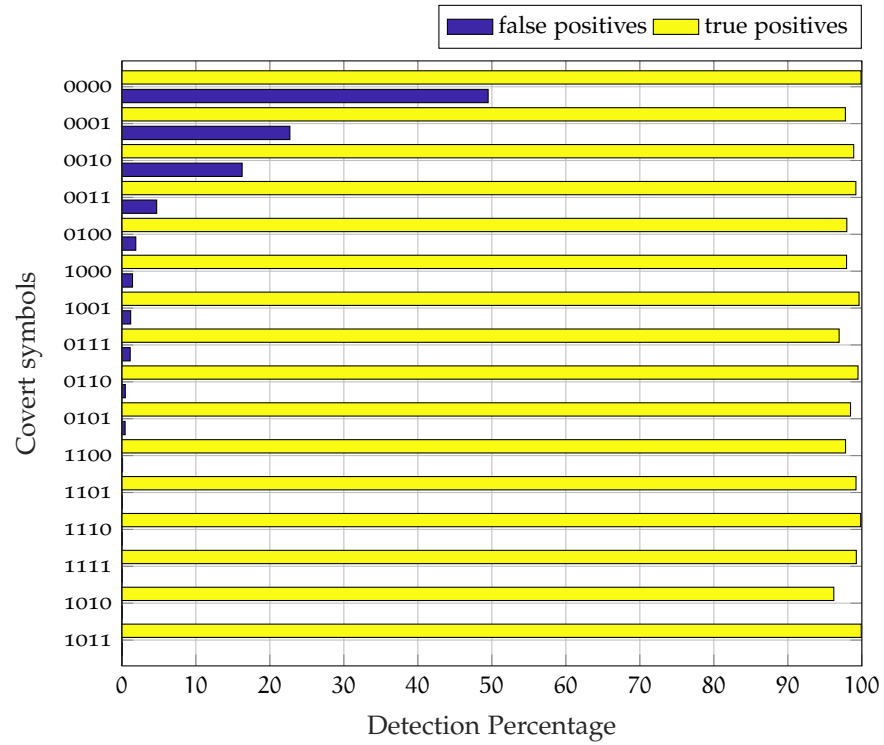


Figure 39: Detection rates at the covert channel receiver. False positives occur when a covert symbol is detected in an unmodified frame. True positives occur when the embedded symbol was detected. (based on [75])

take frame error rates into account. For the other rows below, the influence on regular Wi-Fi receivers increases which renders those symbols inadequate for staying undiscovered.

### 13.3.3 Reception performance at the covert channel receiver

Secondly, we analyze how well the intended receiver of the covert channel can detect and differentiate the symbols. To this end, we analyze the CSI dumps and extract the phases at all subcarriers. As all of our covert symbols are encoded by changing the phases at four subcarriers by roughly  $180^\circ$ , we search for jumps at the expected positions at the receiver's CSI phases and calculate a likeliness value for each symbol to be contained in the received frame. At the end, we choose the symbol with the highest likeliness value as the one we extract for the corresponding frame. If the detected symbol equals the embedded symbol we have a true positive. In Figure 39, we illustrate the percentage of achieving true positives for each symbol. Here, we realize that all different symbols achieve true positive rates reaching 100 percent. Which makes them all usable in a practical covert channel. Additionally, we also analyzed the percentage of false positives. Those are symbols that we detect even though no symbols were em-

*The embedded covert symbols should be extractable (true positives), but symbols should not be found when none were embedded (false positives).*

bedded in the frame. Some symbols are more likely to be detected in this case. In general, even these symbols are usable in practical covert channels, but they should be used as data symbols rather than start and stop symbols of a message. Otherwise, we risk detecting many wrong message starts.

*Symbols with low false positive rates are preferable to avoid ambiguity errors especially for control symbols.*

#### 13.3.4 Choosing suitable symbols

Combining the results of the two measurements, we choose the six symbols  $1111_2$ ,  $1011_2$ ,  $0100_2$ ,  $0010_2$ ,  $1110_2$  and  $0111_2$  as covert channel symbols. As  $1111_2$  does not result in false positives, we choose it as message start symbol.  $1011_2$  is our message stop symbol and the other four symbols are data symbols, each encoding two bits:  $00_2$ ,  $01_2$ ,  $10_2$  and  $11_2$ . For our third experiment, we use the selected symbols to transfer bytes over the covert channel. To signal the receiver that a data transmission starts, we first send the message start symbol and then encode our bits with data symbols followed by a stop message symbol. We chose to place one byte per message, which results in six symbols that need to be transmitted. Of course, the number of bits can be varied to more efficiently use the covert channel as long as both receiver and transmitter use and expect the same fixed number of data symbols per message. To transmit the messages, we embed them in acknowledgement frames that we can generate for our communication partner in advance. Then, we use our ucode modifications to trigger the transmission of the correct acknowledgement corresponding to the covert symbol that should be transmitted. The start pointers for the acknowledgements are stored in the shared memory and written by the ARM core after mapping the desired messages to covert symbols.

*We chose two message control (start and stop) and four data symbols.*

*We embed the chosen symbols into acknowledgements and store the resulting samples in Template RAM.*

#### 13.3.5 Real-time experiments involving the D11 core

In general, our covert channel could communicate normally with a regular Wi-Fi node and embed the covert symbols into the acknowledgements for this node. The covert channel receiver could be another node that eavesdrops on the communication between the two nodes to capture the acknowledgements and dump the CSIs to extract the covert symbols. Using this setup, the covertly communicating nodes could conceal the communication partners. For our experiments, we simplify this setup by using the same node as communication partner for the regular Wi-Fi transmissions and as covert channel receiver. In this setup, the Wi-Fi communication partner simply injects frames to the covert channel transmitter. The latter checks for an expected MAC address and transmits a raw acknowledgement with an embedded covert symbol. The covert channel receiver captures this frame, dumps the CSI and extracts the covert symbol and

*Embedding covert symbols in the communication with a third party hides the intended covert channel receiver.*

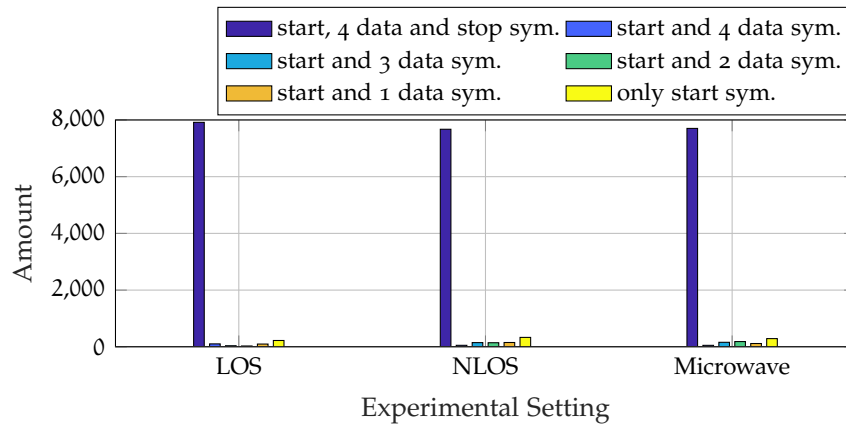


Figure 40: Results of transferring one byte in four covert symbols plus start and stop symbols in a line-of-sight (LOS), non-line-of-sight (NLOS) and microwave environment with changing wireless channel characteristics. (based on [75])

then tries to reconstruct the covert messages, each consisting of one byte.

We ran experiments on our covert channel implementation in three indoor scenarios that we illustrate in Figure 37. First, we keep two Nexus 5 smartphones close together in the line-of-sight setup on the living room table. Then we evaluate a non-line-of-sight setup by keeping one smartphone on the living room table and placing the other on the kitchen countertop so that a direct line-of-sight connection is impeded by a fridge. Both setups simply evaluate the influences of multi-path effects on our covert transmission. In the third setup, we check whether our implementation can cope with quickly changing wireless propagation characteristics, by placing one smartphone in a microwave oven and another one on the kitchen countertop similar to our CSI extraction experiment in Section 8.2. By quickly opening and closing the microwave oven door, we constantly change the wireless channel and thereby the measured CSI.

### 13.3.6 Evaluating the experimental results

In Figure 40, we illustrate the results of the three experiments. For each experiment, we count how many correct symbols in a row we were able to extract leading to either completely correct message receptions, partially correct message receptions or wrong message receptions. In total, we transmitted 10 000 messages. The missing acknowledgements at the receiver were either not correctly received or not transmitted due to a reception problem of the incoming Wi-Fi frame at the covert channel transmitter. The first bar in Figure 40 shows how many messages we extracted completely correct. In all three experiments, most of the extracted covert symbols resulted in complete

*In the first two experiments we evaluate the influence of multi-path effects and in the third, we investigate quickly changing environments.*

*In any of the three setups we successfully received almost 80 percent of the transmitted covert messages, each consisting of 6 acknowledgement transmissions.*

messages. Only few messages were missing symbols to reconstruct the full message. Overall, these results show that our covert channel is indeed practical even in non-line-of-sight environments as well as those with constantly changing wireless propagation characteristics due to movement in the environment (in this case the microwave oven door). Further error correction of the received messages is left for upper layers that might even enhance confidentiality of the transmitted covert symbols by applying cryptography with a shared key between the covert channel communication partners. This would at least hinder leaking the covertly transmitted information in case an adversary discovers and decodes this covert channel.

*To change wireless propagation characteristics over time, we moved a microwave oven door.*

## 13.4 DISCUSSION

To the best of our knowledge, we developed the first physical-layer covert channel where both the transmitter and the receiver are implemented on an off-the-shelf Wi-Fi chip that is even installed in smartphones. Especially for people living in countries where information flows are strictly regulated, covert channels such as ours can help to exchange messages between wirelessly enabled users without even revealing that a covert transmission is ongoing as well as who the receiver of the messages is. On the other side, adversaries can also use covert channels to secretly extract private information from a protected system. Even if intrusion detection systems may analyze the regular Wi-Fi frame contents, they most likely miss irregularities on the physical layer. To counter these information leakage attacks, security experts need to know covert channel implementations to be able to at least detect the existence of an information leak so that it can be found and closed. Hence, for both applications, it is important to discuss new ways of implementing covert channels.

*We developed the first physical-layer covert channel fully implemented on smartphones.*

## 13.5 RELATED WORK

The idea of hiding information in wireless network traffic is not new. Covert channels use properties of legitimate communication channels to transmit additional data that is invisible to uninitiated receivers. The term was first used by Lampson in 1973 [54].

*The term covert channel exists since 1973.*

### 13.5.1 Data-link-layer approaches

Most covert channels are designed for the data-link layer or higher layers. They use reserved fields, time delays, or packet corruptions to embed covert information into a data stream. In [29, 93], Girling and Wolf introduce covert channels for local area networks (LANs). In [38], proposed an approach for transmitting data by corrupting Wi-Fi frames. The covert information is either stored in the WEP ci-

*Most covert channels on Wi-Fi's data-link layer concentrate on overwriting parts of the Wi-Fi headers.*

pher's initialization vectors (IVs), MAC addresses or the frame check sequence (FCS). In [28], Frikha and Trabelsi additionally consider the sequence control field in the Wi-Fi headers to store hidden bits. In [62], Martins and Guyennet propose the use of reserved fields in 802.15.4 systems as covert channels. In [52], Krätzer et al. present an analysis of campus traffic and evaluated utilizable fields for hiding information based on randomness and occurrence of set bits.

### 13.5.2 Physical-layer approaches

*Of the few existing physical-layer covert channels, even less are evaluated in practical setups.*

*Covert channels for OFDM-based systems can be reused in multiple wireless communication standards.*

Wireless physical-layer covert channels are rare, but they are more generic as technologies such as OFDM are used in various wireless standards. In [18], we present and evaluate four covert channels for Wi-Fi systems in the field. Two were first presented in this work. The STF PSK covert channel shifts the phase of STF symbols which are found at the beginning of a Wi-Fi frame's preamble. The position makes this covert channel resistant against reactive jamming that does not target all Wi-Fi traffic. The CFO FSK covert channel introduces frequency offsets into each OFDM symbol imitating very quickly changing Doppler effects on Wi-Fi frames. The Camouflage Subcarrier covert channel was first presented in a similar form by Hijaz and Frost in [39]. While they focus on adding covert-information-carrying subcarriers to LTE and WiMAX systems in simulation, we added those to Wi-Fi signals and evaluated the performance in practical experiments. The Cyclic Prefix Replacement covert channel from [18] was previously proposed by Grabski and Szczypiorski in [32]. This channel places covert symbols into the cyclic prefix that is required to cope with inter-symbol interference (ISI) due to multi-path propagation. While Grabski and Szczypiorski only simulated with AWGN channels, we performed practical experiments and simulations in multi-path environments. In [24], Dutta et al. present an alternative approach called Dirty Constellations. It embeds covert symbols by adding small IQ symbols to existing constellation points. This is observed as additional noise and, therefore, only applicable to links that do operate close to the channel capacity limits.

### 13.6 FUTURE WORK

*The flexibility of the Nexmon SDR allows implementations of various practical physical-layer covert channels on mobile devices.*

Our filter-based covert channel is only one out of many covert channels implementable for Wi-Fi systems on the physical layer. Due to the flexibility of the SDR-based transmission approach, we are very flexible when it comes to modifying Wi-Fi frames. As long as real-time operation is not an issue or frames with covert symbols can be generated in advance, any of the covert channels from the related work section should be implementable. In addition, also covert channels for other communications standards are possible. Leading to the

new concept of cross-technology covert channels. As Broadcom chips installed in smartphones generally integrate both a Wi-Fi and a Bluetooth transceiver connected to the same antenna, investigating covert channels for Bluetooth becomes possible. With 80 MHz bandwidth available in the 2.4 GHz band, we cover the complete Bluetooth band.

For various applications, it is sufficient to only implement the transmitter of a covert channel in an off-the-shelf device. More flexible equipment such as software-defined radios can be used to receive and decode the covert frames. Nevertheless, the ability to also use Wi-Fi chips to capture raw samples of radio transmissions allows us to capture most physical-layer covert channels in the Wi-Fi bands. However, frame detection and synchronization, especially for non-Wi-Fi transmissions, are computationally expensive operations. The reception of Wi-Fi-based covert channels could at least be triggered by existing frame detection hardware build into the Wi-Fi chip to avoid capturing raw samples all the time. In future work, we should investigate physical-layer properties that can be extracted with off-the-shelf receivers to then build covert channels that use those properties to hide information in Wi-Fi frames.

*Covert channel receivers should use existing hardware components to extract physical-layer properties whenever possible to avoid otherwise required continuous raw sample captures.*

### 13.7 CONCLUSION

In this chapter, we introduced a novel covert channel for Wi-Fi systems that imitates effects of fading channels to encode covert symbols into Wi-Fi frames. To this end, we use transmit filters that modify phases on selected subcarriers to represent covert symbols. A covert channel receiver can extract those symbols by analyzing the channel state information that regular receivers use to equalize the fading effects of the wireless channel. We implemented both our covert channel transmitter and the receiver in the Wi-Fi chip of Nexus 5 smartphones and evaluated their performance in an apartment. As we used the Nexmon SDR presented in Chapter 9 to transmit frames with embedded covert information, we focused on transmitting pre-generated acknowledgement frames including different covert symbols to reduce delays in real-time communication systems. Our results show, that regular receivers are only disturbed by embedded covert symbols when high modulation coding schemes are in use. The covert channel receiver has very good detection rates and can extract covert messages that span over multiple Wi-Fi frames.

*We demonstrated the practicability of our transmit-filter-based covert channel under real-time conditions in realistic environments with changing wireless channels.*

### 13.8 MY CONTRIBUTION AND ACKNOWLEDGEMENTS

Implementing this covert channel on smartphones was my idea and the initial motivation to even start reverse engineering Wi-Fi firmware. I thank Francesco Gringoli for the intensive collaboration especially for writing the ucode that implements the frame-triggered acknowl-

*A Covert channel on smartphones was the spark that ignited the interest in modifying Wi-Fi firmwares.*

edgement transmissions. I also thank Jakob Link for implementing a first version of the presented covert channel for his bachelor thesis [59] and extending it for this work. I also thank Jiska Classen for collecting the related work on covert channels. With her, I started the first investigations on Wi-Fi covert channels presented in [18].



## PROJECTS USING NEXMON

---

We developed Nexmon as an open-source project, so that others may not only benefit from our results, but also reuse our code in their own projects. In this chapter, we present projects that are directly based on Nexmon or that partially use information published in our project. In many projects, we participated in either porting Nexmon to a different platform or we performed reverse engineering tasks required to achieve the goals of collaborating groups.

### 14.1 NEXMON FOR QUALCOMM'S 802.11AD WI-FI CHIP

Even though Nexmon focuses on Broadcom Wi-Fi chips, the framework can generally be used for any microcontroller architecture that is supported by the GNU Compiler Collection (GCC) with enabled plugin support. An analysis by D. Steinmetzer and D. Wegemer of Qualcomm's QCA9500 802.11ad Wi-Fi chip revealed that the chip contains two ARC600 processors. One used for real-time operations (ucode processor) and another for interfacing the host system and performing FullMAC operations (firmware processor). Based on their findings, we started a collaboration to port the Nexmon framework to support the QCA9500 chip.

*With Nexmon we support Qualcomm's first commercially available 802.11ad Wi-Fi chips installed in home routers.*

#### 14.1.1 Porting Nexmon to ARC600 cores

Compared to ARM processors, the ARC600 is based on a Harvard architecture with separate instruction and data memory regions. As illustrated in Figure 41, the individual instruction memory regions seen by each of the two cores start at address 0x0 and are write protected. The data regions start at address 0x80000 and are writable. Traditionally, the Nexmon approach requires that data and instruction regions are combined and, hence, both writable. To port the framework, we had to either extend Nexmon to support separate instruction and data regions or to find a way to get write access to the instruction regions. During our analysis, we realized that the driver loads firmware binaries into the Wi-Fi chip by writing to addresses above 0x8c0000. From there, the memory regions are mapped to the cores' instruction and data address spaces as illustrated in Figure 41. As a check whether the internal cores could also write to addresses above 0x8c0000 succeeded, we decided to modify the Nexmon framework to merge both instruction and data sections into the code memory partitions above 0x8c0000. We instructed the linker to

*Obtaining write access to code memory on Harvard architectures eases the portation of Nexmon to such platforms.*

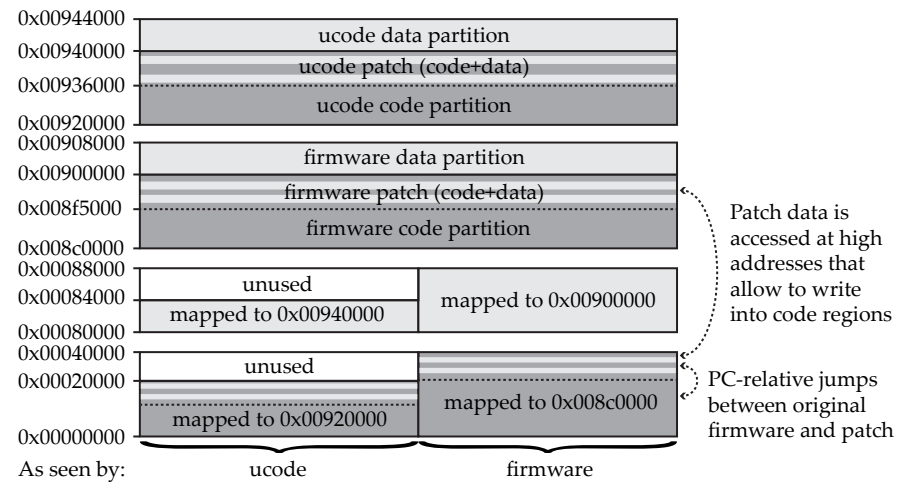


Figure 41: Memory layout of the QCA9500 802.11ad Wi-Fi chip with two ARC600 processors (ucode and firmware) that have separate write-protected code and writable data memories at low addresses. All four memory regions are remapped into high addresses, where they are writable and accessible from the host. (based on [85])

assume that the code will be loaded into addresses starting from 0x0 during runtime, so that the linker produces program counter relative jump instructions to jump between patch and original firmware code. For writing to data regions, we pointed the linker to addresses above 0x8c0000 as they are writable by the internal ARC600 cores. These changes to the Nexmon framework allowed us to use it with the QCA9500 chip.

#### 14.1.2 Simplifying debugging of the QCA9500 firmware

*Adding a printf function to the firmware simplified runtime analyzes of the firmware.*

Compared to the Broadcom firmwares, the QCA9500 firmware was much harder to reverse engineer. Especially, as the firmware does not come with an internal console to which the developers may print debug messages that would have helped us to indicate the purpose of the functions generating those messages. To dynamically analyze the firmware during runtime, we created an internal console to which strings may be written by calling a printf function. We also extended the driver to dump this console into the user space of the host's operating system.

Equipped with such tools, D. Steinmetzer and D. Wegemer were able to analyze and extend the QCA9500 firmware to, for example, enhance the sector selection algorithm required to optimally communicate between 802.11ad stations in the 60GHz band. This work resulted in the joint publication [85].

#### 14.2 NEXMON FOR FITBIT ACTIVITY TRACKERS

Even though Nexmon started as a patching framework for Wi-Fi firmwares, it also allows to create modifications for other devices such as Fitbit activity trackers. These devices are also based on ARM processors which simplifies the adaptation of our framework to this platform. In [19], J. Classen and D. Wegemer demonstrate how to use Nexmon to add new features to the Fitbit firmware to simplify dynamic firmware analyses.

*Porting Nexmon to other ARM-based devices is simple.*

#### 14.3 SECURITY ANALYSES BASED ON NEXMON'S RESULTS

In the year 2017, two remote code execution vulnerabilities in Broadcom Wi-Fi firmware have been discovered. The first, discovered by G. Beniamini in [7], exploits a buffer overflow in a vendor extension of the tunneled direct link setup (TDLS) protocol implementation and can be triggered by any device that is connected to the same wireless network as the target node. The second, discovered by N. Arstenstein in [3] exploits a buffer overflow in the code handling the wireless multimedia extensions (WME) field in probe response frames. It can be triggered by simply replying to a probe request frame sent by the target node.

*Multiple remote code execution vulnerabilities were found in Broadcom Wi-Fi firmwares.*

Both projects reference our Nexmon project [80] as one reference to start the analysis of the firmware. Hence, our project helped in finding and fixing new security vulnerabilities. Unfortunately, not all smartphone vendors offer Wi-Fi firmware updates for all their affected devices, so that many devices will stay vulnerable. At least, Nexmon allows to patch the discovered security holes even for legacy devices. To enhance the security of wireless communication systems in general, it would be helpful to force chip vendors by law to publish their firmware source code so that security holes can be found and patched more easily without having to rely on the support of the chip vendor.

*Open source firmwares would simplify firmware patches on legacy devices.*

#### 14.4 NEXMON FOR QUALCOMM'S LTE MODEM FIRMWARE

In his master thesis [13], C. Bruns analyzed LTE modem firmware used in MSM8974 chips and created a patching framework based on Nexmon's firmware patching concept. Compared to Broadcom's Wi-Fi firmwares, Qualcomm's LTE modem firmwares are generally verified by the chip's bootloader. Only correctly signed firmwares should be executable. For development, however, special publicly available certificates can be used to sign firmwares. Alternatively, the developers can also completely disable firmware verifications. Bruns found the hint, that some vendors even sell devices with such developer

*Qualcomm employs a working firmware verification solutions that we can only avoid on devices that are misconfigured.*

options enabled. Hence, he was able to modify the LTE firmware running on the Asus PadFone Infinity 2.

*Firmware patches lead to various new applications running in the LTE modem.*

To this end, he had to generate code for Qualcomm's Hexagon digital signal processor that runs the firmware on the LTE chip. Similar to Nexmon's approach, Bruns also stores additional placement information into C files containing the patch code and can thereby overwrite and extend existing code. His example applications include a sniffer for data-link-layer frames, a channel state information (CSI) extractor and an encryption key extractor. All these features are not officially offered by the original firmware so that Bruns was the first to develop these firmware extensions for the research community.

#### 14.5 CONCLUSION

*Nexmon is a versatile approach to comfortably modify firmware of various devices.*

The projects presented above are an example for the flexibility of the Nexmon firmware patching approach. The idea of modifying firmware by writing C code is not limited to Broadcom chips, but can be applied to many other platforms. Our results prove that Nexmon has the capabilities to become the standard tool for modifying binary firmware that consists of proprietary code running on integrated communication chips, Internet-of-things devices or other embedded systems.

## Part V

### DISCUSSION AND CONCLUSIONS

We discuss our findings and applications in Chapter 15.  
Then, we conclude this thesis in Chapter 16.



## DISCUSSION

Before we started working on Nexmon, we faced the problem of having only limited control over Wi-Fi chips installed in smartphones. Generally, those chips act as Ethernet-to-Wi-Fi bridges that handle all frame processing steps and the interaction with the wireless interface in a proprietary firmware—that equals a blackbox. In such a system, only the chip’s manufacturer and maybe the manufacturers of devices using such chips have access to the firmware source code and are able to develop new applications. This gives companies who can afford to cooperate with chip manufacturers a monopoly to drive innovation. Apple, for example, performs distance bounding as an option to unlock computers. The implementation relies on distance measurements collected by custom Wi-Fi firmware extensions. The lack of freely available firmware source code and proper chip documentation hampers researchers to present similar solutions and make them available to a wider public. Additionally, end users heavily rely on the manufacturers’ good will to fix security vulnerabilities in firmwares. Access to firmware source code would allow trusted third parties to update firmware even if the manufacturer drops support for a particular device.

While access to the source code would be the royal road to new Wi-Fi based applications, it seems unlikely that manufacturers would be that open. In the last decade they already moved back from rather open SoftMAC Wi-Fi cards to FullMAC Wi-Fi cards running proprietary firmware. With Nexmon, we supply a solution, that might become obsolete as soon as open firmware becomes available. Until then, Nexmon is the only way to efficiently analyze, modify and extend Wi-Fi firmware of Broadcom’s FullMAC chips. And it is by far not limited to such devices as Nexmon-based projects for Qualcomm’s 802.11ad devices and FitBit’s activity trackers show. The ability to write firmware patches in C or even embed existing C code into firmware patches gives researchers a flexibility close to having direct access to the firmware source code. Nevertheless, they still have to reverse engineer the firmware binary first to find appropriate places to branch execution into their new code. This is the challenge only a few researchers are willing to face.

When we started our research on Broadcom Wi-Fi chips, our ambitious goal was the implementation and evaluation of a physical-layer covert channel on smartphones. To achieve this goal, we first had to understand how the Wi-Fi firmware operates in general. The first side product of this research was the implementation of a patch that

*Proprietary firmware renders Wi-Fi chips into blackboxes that bridge between Ethernet interfaces and Wi-Fi networks.*

*Creating innovations can be simplified by giving researchers access to firmware source code and proper hardware documentation.*

*Nexmon is only required where access to source code is restricted—which is often the case.*

*While Nexmon simplifies the creation of patches, reverse engineering firmwares is a required groundwork.*

*Nexmon was initiated to build covert channels.*

*On the stony way to covert channels, we created side products beneficial for various applications.*

*SDRs are very flexible but also require computationally expensive generations of raw signals.*

*Nexmon does not require a deep understanding of Wi-Fi firmware to get started, instead, beginners can simply use the tools we created and published.*

activates monitor mode and frame injection on various smartphones. Based on what we learned about the Wi-Fi chip, we were able to evaluate the energy-wise and delay-wise benefits of moving frame processing steps into the firmware, which also helped us improving software-defined wireless networking applications. After that, we were still left with the tasks of embedding covert symbols into Wi-Fi frames and extracting them on the receiver side. For the latter, we already knew that it was possible to extract channel state information (CSI) on various Wi-Fi chips, but we still had to understand how to access and interpret this information on smartphone Wi-Fi chips. The resulting channel state information extractor is a versatile tool that can be reused by other researchers who require this information especially on mobile devices. For embedding the covert symbols, we first intended to use beam-steering capabilities of multi-antenna Wi-Fi chips. But before spending time into this direction, we discovered software-defined radio (SDR) capabilities. They are not specialized for specific operations and thereby less efficient. Nevertheless, SDR capabilities offer maximum flexibility for transmitting arbitrary signals in the Wi-Fi bands. Discovering this feature is a breakthrough for communicating in different communication standards using off-the-shelf Wi-Fi devices. This feature also added the little extra to our reactive Wi-Fi jammer as it is now possible to transmit arbitrary jamming signals from a smartphone. Both SDR and CSI extraction capabilities allowed us to achieve our initial goal of implementing and evaluating a physical-layer covert channel on smartphones. Reaching this goal required perseverance and the urge to continuously discover some new functionality that could be used to implement ideas of novel applications.

Our future work sections concluding various chapters show, that the investigation into Wi-Fi chips is by far not finished and the possibilities for developing novel Wi-Fi-based applications for smartphones are not yet exhausted. Due to the open source mindset we propagate in our projects, other researchers can easily get started with firmware modifications to enhance their own projects and evaluate their solutions in real-world testbeds with mobile end devices. Just using new capabilities such as monitor mode, frame injection, reactive jamming, CSI extraction or SDR-based transmissions, beginners do not even need to understand the chip internals—not even how the firmware works. They can use tools we provide and build their applications on top. After getting started, they can dive into firmware patching and generate first extensions on the frame processing path or adjust the behavior of existing patches according to their requirements. Only if functionalities are required that exceed the existing set of reverse engineered functions, structures and hooking addresses, beginners have to become advanced Nexmon users and look into the firmware binary on their own to find the missing pieces. This opens the way of



finding and reverse engineering new capabilities such as beam steering matrices that define how multiple spatial streams are transmitted in multi-antenna systems. We are curious to know what others may create based on Nexmon.

*New exciting applications based on Nexmon will emerge.*



## CONCLUSIONS

In this work, we evaluated how smartphone-based wireless systems can be enhanced performance-wise and security-wise. To this end, we reverse engineered Broadcom's FullMAC Wi-Fi firmware as well as chip internals and developed a framework to write firmware patches in C and automate the patching process. Of course, running modified firmware created by third parties entails risks. Malicious entities could write firmware patches that spy on a user's communication or turn its device into a communication jammer. To protect users from such attacks, we proposed to enhance Broadcom's firmware loading mechanism by one that verifies that firmwares are signed by a trusted entity. In case researchers want to load custom firmwares onto their own Wi-Fi chips, we proposed means to replace the public signature verification key by having physical access to the Wi-Fi chip itself. To make sure that users can verify that our released firmware patches are free of—unintentional—malicious functionalities, we released their source code. Users may verify our implementation, reproduce our experiments and extend our code to use it in their own applications. To demonstrate the capabilities of modern Wi-Fi chips, we created and evaluated various example applications that even exceed the capabilities one would expect from Wi-Fi chips in off-the-shelf smartphones.

The activation of monitor mode and frame injection on smartphone Wi-Fi chips gives researchers easy access to raw frames. This allows to evaluate prototypes of new communication paradigms within the host's operating system. To reduce energy consumption and frame processing delays, parts of such implementations can be offloaded into the Wi-Fi chip. This is especially beneficial for frame forwarding applications used in wireless mesh networks. Implementing those on smartphones creates a versatile and realistic mobile evaluation platform. We demonstrated the advantages of answering pings in the Wi-Fi firmware compared to answering them in the operating system's kernel in our ping-offloading application—energy consumption and round trip times are significantly reduced. Firmware implementations can even enhance software-defined wireless networks. While we focused on controlling the physical-layer based on application requirements using WARP software-defined radios connected to smartphones, a Nexmon-capable Wi-Fi chip can fulfill similar goals without additional hardware components besides a smartphone. In our example application, we evaluated how the performance of broadcasting videos encoded by scalable-video codecs can be enhanced by adjusting modulation coding schemes and transmission powers per video

*We reverse engineered Wi-Fi firmwares and developed a patching framework.*

*Firmware modifications can contain malicious code that can compromise the security of a device.*

*Monitor mode and frame injection is the first step to get better access to the wireless channel.*

*Offloading functionalities into the firmware can save energy and reduce the processing delays.*

*Friendly jammers  
can be used to  
enhance the security  
of a network by  
destroying malicious  
frames.*

*Using off-the-shelf  
devices as jammers  
can lead to  
wide-spread  
malicious attacks  
against wireless  
networks.*

*Our physical-layer  
covert channel  
allows to exchange  
information without  
revealing to an  
eavesdropper that a  
communication link  
exists.*

*Research into covert  
channel is a  
double-edged sword  
as disclosing how  
they work likely  
helps to detect and  
uncover them.*

quality layer. Instead of directly defining those parameters within the application, we used the concept of flows to which we assign transmission requirements that are translated into physical-layer transmission settings in a service that has an overview over all flows on a device.

Focusing on the security enhancements, we demonstrated that it is possible to implement reactive Wi-Fi jammers on smartphones. To this end, we manipulated the firmware running on the chip's real-time processor. It can inspect frames during reception and trigger the transmission of a jamming signal in case a jamming condition matches. We even proposed and implemented two novel jamming enhancements. One enhancement makes sure that while destroying the reception of targeted flows, transmitters of such flows continue their non-targeted communication. The other enhancement evaluates jamming successes to minimize the jammer's power consumption. On the one hand, we intend to use this jammer for friendly jamming purposes. Those could be physical-layer firewalls that can destroy malicious frames before they can be received by otherwise unprotected devices. On the other hand, we also want to make other researchers aware of security implications that rise from the omni-presence of wirelessly connected devices. Malicious attackers could take over a significant number of smartphones to launch a massive reactive jamming attack against our wireless infrastructure. While this attack can never be completely avoided, the restriction to only loading signed firmwares onto communication chips reduces the attack surface.

Our second security-related application focuses on the protection of people who intend to exchange information without disclosing that this exchange even occurred. To this end, we implemented a physical-layer-based covert channel for Wi-Fi systems on smartphones. It embeds covert symbols into phase changes on selected subcarriers in OFDM-based Wi-Fi transmissions. To extract the embedded symbols, we extract channel state information at a receiving smartphone. It contains phase and amplitude changes introduced by the wireless channel as well as those introduced by the transmitter. Similar to jammers, also covert channels are dual-use applications. They can help informants to share their information with journalists but they can also help spies to covertly extract information from computer systems. The latter scenario requires the research on covert channel detection and counter mechanisms. However, those developments would also endanger the people in the first usage scenario. Nevertheless, only by openly discussing possible covert channel implementations and defenses against them, one can assess the risks of using them in practice.

We based the implementation of the presented applications on our discoveries of undocumented capabilities of Wi-Fi chips. The first is the extraction of channel state information on smartphones, the

second is the Wi-Fi chip's ability to transmit and capture raw base-band signals. This way, off-the-shelf smartphones can be operated like software-defined radios. We offer tools to experiment with both discoveries and use them for future research applications. To additionally help researchers to find and understand currently unexplored parts of the Wi-Fi firmware, we also created a programmable debugger that supports single stepping and directly integrates into the firmware. We hope that the presented tools and applications inspired other researchers to further dive into firmware analyses to unleash the full potential of commercial off-the-shelf hardware.

Overall, this thesis proves that many applications previously requiring software-defined radios for their implementation and evaluation, can now be implemented on relatively cheap off-the-shelf smartphones. This allows for large-scale testbeds that support mobility and already come with a set of example applications other researchers can build on.

*Software-defined radio, CSI extraction and debugging projects give other researchers a starting point for working with Nexmon.*

*Nexmon enhances smartphone Wi-Fi chips to benefit large-scale, mobile, communication testbeds.*



Part VI

APPENDIX







## SOFTWARE RELEASES

---

Writing this thesis, we created various tools and applications as well as the Nexmon framework itself. To maximize the benefits for the community, we published our developments as open-source software. This simplifies other researchers to reproduce our results and develop their own applications based on our tools. In the following sections, we present our software releases.

*Open-source software simplifies reproducibility and the extension of our patches.*

### A.1 NEXMON FIRMWARE PATCHING FRAMEWORK

The foundation for this work is the Nexmon firmware patching framework. It contains tools for handling and patching firmware binaries. We keep them in the `buildtools` directory in the repository mentioned below. It contains a precompiled GCC compiler toolchain for ARM microcontrollers to compile C files into binaries. Our `gcc-nexmon` plugin extends the GCC compiler to handle `at`-attributes and `targetregion`-pragmas that allow to define where symbols should be placed during patching. In the `firmwares` directory, we collect original firmware binaries and store addresses required for the patching workflow. The patches extending those firmwares are stored in the `patches` directory. They consist of various C files containing code and placement information and a `Makefile` that describes the process of building patches and integrating them into the original firmware binaries. For most firmwares, we supply a `nexmon` example project that enables monitor mode with `radiotap` headers and frame injection capabilities. It is a good starting point for implementing new patches. While most smartphones do not require driver modifications, the `brcmfmac` driver used on the Raspberry Pi misses some essential functionality to communicate with the firmware. Hence, we integrated it into the patch directories. On the target platforms, various command line utilities and libraries are required for controlling the firmware (`nexutil` and `dhdutil`) and imitate a monitor interface (`libnexmon.so`) to unmodified analysis and penetration testing applications (`tcpdump` or `aircrack-ng`) that we supply in the `utilities` directory.

*The framework contains everything to build and install firmware patches on smartphones and Raspberry Pis.*

*Besides firmware patches, we supply drivers and command line utilities for control and penetration testing.*

The repository of the Nexmon firmware patching framework is available at:

**<https://nexmon.org/>**

In the following sections, we present firmware patches that can be copied into the framework's patches directory for building.

## A.2 PING-OFFLOADING APPLICATION

*The ping application demonstrates how to handle Ethernet frames within the firmware.*

Our ping-offloading application is a simple patch to demonstrate how to handle Ethernet frames in the firmware. To this end, we hook the `wl_arp_recv_proc` function call in the `ping.c` file to intercept Ethernet frames before sending them to the host and use the `wlc_sendpkt` function to inject new Ethernet frames for transmission. To reproduce the experiments described in Chapter 10, we supply `Makefile.exp*` files. They initiate an ad hoc connection between two Nexus 5 smartphones and start the ping application.

The ping-offloading application is available at:  
**[https://nexmon.org/ping\\_offloading](https://nexmon.org/ping_offloading)**

## A.3 NEXMON DEBUGGER

*The debugger consists of a common file for all debugging applications and an application specific file.*

Our debugger application presented in Chapter 7, mainly consists of the two files `debugger.c` and `debugger_base.c`. The first file is used to set breakpoints and watchpoints and to handle their occurrence. The second file contains all the patches required for activating the debugging functionality in a firmware. It initializes a new stack for handling debugging exceptions, saves the register state to this stack when a debugging exception occurs and calls the corresponding handlers. To use the debugger in another project, one simply needs to copy the two files and adjust the `debugger.c` file according to the required debugging logic.

The debugger application is available at:  
**<https://nexmon.org/debugger>**

## A.4 NEXMON JAMMER

*Due to ucode changes, the jamming application is more complex than the applications presented above.*

More sophisticated is our reactive jamming application, as it requires to change the ucode running on the real-time processor. To avoid publishing the disassembled original code, we placed our changes in a patch file that can be applied to a freshly disassembled firmware file. The `Makefile` extracts the ucode from the ARM firmware file and applies the ucode patches automatically. To decide which of the supplied ucode modifications should be loaded, one needs to pass the `UCODEFILE` parameter when calling `make`. Examples are provided in the `Makefile.exp*` files we used to start the experiments for generating the results presented in Chapter 12. We also provide the source code for our Jamming app as well as the associated firmware, which

is a clean starting point for new projects based on our reactive jammer. To control the jammer and generate jamming signals using the app, we use `ioctl`s starting at command number 500. The file `ioctl_5xx.c` contains the corresponding handlers.

The reactive jamming firmware used to create the presented results is available at:

**<https://nexmon.org/jammer>**

The source code of the Jamming app is available at:

**[https://nexmon.org/jammer\\_demo\\_app](https://nexmon.org/jammer_demo_app)**

The corresponding firmware for the Jamming app is available at:

**[https://nexmon.org/jammer\\_demo\\_firmware](https://nexmon.org/jammer_demo_firmware)**

## A.5 NEXMON SDR

Our software-defined radio application demonstrates how to trigger the transmission of a Wi-Fi acknowledgement frame stored as IQ samples in Template RAM. To generate the frames waveform, we use a MATLAB script. It generates samples in a format expected for raw transmissions with the Wi-Fi chip. To load those samples into the Template RAM, we call `ioctl` 711 defined in `ioctl_7xx.c`. Our MATLAB script generates a shell script that calls this `ioctl` using `nexutil`. After copying this file to a smartphone and executing it, the samples are loaded into the Wi-Fi chip. To start a continuous transmission of the samples stored in Template RAM, we use `ioctl` 770 and to stop the transmission, we use `ioctl` 771. Using a regular Wi-Fi receiver running in monitor mode, we can capture the transmitted Wi-Fi frames (e.g., by using Wireshark).

*We generate samples of a Wi-Fi frame in MATLAB, load them into the Wi-Fi chip's Template RAM and trigger a transmission.*

The software-defined radio application is available at:

**<https://nexmon.org/sdr>**

## A.6 NEXMON CSI EXTRACTOR

Similar to our jamming application, also the channel state information (CSI) extractor requires the modification of ucode. Again, we only included patch files that can be automatically applied to disassembled ucode. As CSI extraction hinders our receiver from correctly decoding the frame payload, we only trigger the CSI extraction if a frame contains the destination MAC address "CSIEXT". To change this, one needs to modify the corresponding code in the ucode file marked with the comment "MAC COMPARISON". During extraction, CSI is first copied from physical-layer tables into shared memory and then pushed to the ARM's RAM by using multiple DMA

*Our CSI extractor targets only frames with destination MAC "CSIEXT" and uses UDP datagrams to pass the extracted information to Android apps.*

transfers. A state machine in the `csi_extraction.c` file handles the reassembly of the whole CSI and packs it into a UDP datagram sent to port 5500 using the broadcast IP address 255.255.255.255. This way, even Android apps can receive CSI dumps for implementing advanced applications.

The channel state information extraction application is available at:  
**[https://nexmon.org/csi\\_extractor](https://nexmon.org/csi_extractor)**

#### A.7 NEXMON COVERT CHANNEL

*The transmitter pre-filters outgoing frames to embed covert symbols and the receiver extracts them from CSI dumps.*

The covert channel application combines both the software-defined radio and the CSI extractor applications. It uses SDR transmission capabilities to send raw samples of Wi-Fi frames that have been pre-filtered in the time-domain to embed covert symbols. As described in Chapter 13, we send whole bytes split over three frames. Two contain start and stop symbols. The other four contain symbols representing two bits each. At the receiver's side, we use CSI dumps to detect which symbols were embedded into the Wi-Fi frames. To this end, we have a MATLAB script that compares the CSI dumps with the embeddable symbols to find the most likely ones.

The channel state information extraction application is available at:  
**[https://nexmon.org/covert\\_channel](https://nexmon.org/covert_channel)**

#### A.8 NO-LTE KERNEL FOR NEXUS 5

*To avoid disturbing peaks in power measurements, we disabled LTE-related hardware.*

For measuring energy consumption of Nexus 5 smartphones, we used a Monsoon Power Monitor. It connects to the phone's battery ports to act as the only power source connected to the smartphone. When measuring power consumption while the phone is idle, we observed regularly reappearing peaks in the power measurements, which may falsify our measurements. We realized that the peaks are caused by LTE-related hardware that can be disabled by turning off the `CONFIG_MSM_SMD_PKT` setting when building the kernel.

The kernel source code used in our experiments is available at:  
**[https://nexmon.org/energy\\_measurement](https://nexmon.org/energy_measurement)**

## BIBLIOGRAPHY

---

- [1] K. Ali, A. X. Liu, W. Wang, and M. Shahzad. "Recognizing Keystrokes Using WiFi Devices." In: *IEEE Journal on Selected Areas in Communications* 35.5 (May 2017), pp. 1175–1190.
- [2] N. Anand, S.-J. Lee, and E. W. Knightly. "STROBE: Actively Securing Wireless Communications using Zero-Forcing Beamforming." In: *Proc. of the 31st IEEE International Conference on Computer Communications*. INFOCOM '12. IEEE, 2012, pp. 720–728.
- [3] N. Arstenstein. *Broadpwn: Remotely compromising Android and IOS via a Bug in Broadcom's Wi-Fi Chipsets*. July 2017. URL: <https://blog.exodusintel.com/2017/07/26/broadpwn/>.
- [4] I. E. Bagci, U. Roedig, I. Martinovic, M. Schulz, and M. Hollick. "Using Channel State Information for Tamper Detection in the Internet of Things." In: *Proc. of the 31st Annual Computer Security Applications Conference*. ACSAC '15. ACM, 2015, pp. 131–140.
- [5] M. Bansal, J. Mehlman, S. Katti, and P. Levis. "OpenRadio: A Programmable Wireless Dataplane." In: *Proc. of the First Workshop on Hot Topics in Software Defined Networks*. HotSDN '12. ACM, 2012, pp. 109–114.
- [6] E. Bayraktaroglu, C. King, X. Liu, G. Noubir, R. Rajaraman, and B. Thapa. "On the Performance of IEEE 802.11 under Jamming." In: *Proc. of the 27th IEEE International Conference on Computer Communications*. INFOCOM '08. IEEE, 2008, pp. 1265–1273.
- [7] G. Beniamini. *Over The Air: Exploiting Broadcom's Wi-Fi Stack (Part 1)*. 2017. URL: [https://googleprojectzero.blogspot.de/2017/04/over-air-exploiting-broadcoms-wi-fi\\_4.html](https://googleprojectzero.blogspot.de/2017/04/over-air-exploiting-broadcoms-wi-fi_4.html).
- [8] D. S. Berger, F. Gringoli, N. Facchi, and I. Martinovic. "Gaining Insight on Friendly Jamming in a Real-world IEEE 802.11 Network." In: *Proc. of the 7th ACM Conference on Security and Privacy in Wireless & Mobile Networks*. WiSec '14. ACM, 2014, pp. 105–116.
- [9] D. S. Berger, F. Gringoli, N. Facchi, I. Martinovic, and J. B. Schmitt. "Friendly Jamming on Access Points: Analysis and Real-World Measurements." In: *IEEE Transactions on Wireless Communications* 15.9 (2016), pp. 6189–6202.
- [10] A. Blanco and M. Eissler. *MonMob project repository*. 2012. URL: <https://github.com/tuter/monmob>.
- [11] A. Blanco and M. Eissler. *One firmware to monitor 'em all*. 2012. URL: <http://www.coresecurity.com/corelabs-research/publications/one-firmware-monitor-em-all>.
- [12] J. Brown, I. E. Bagci, A. King, and U. Roedig. "Defend Your Home!: Jamming Unsolicited Messages in the Smart Home." In: *Proceedings of the 2nd ACM Workshop on Hot Topics on Wireless Network Security and Privacy*. HotWiSec '13. ACM, 2013, pp. 1–6.

- [13] C. Bruns. "Modification of LTE Firmwares on Smartphones." M.Sc. thesis. Technische Universität Darmstadt, Germany, 2017.
- [14] Y. Cai, K. Xu, Y. Mo, B. Wang, and M. Zhou. "Improving WLAN Throughput via Reactive Jamming in the Presence of Hidden Terminals." In: *Proc. of the 2013 IEEE Wireless Communications and Networking Conference*. WCNC '13. IEEE, 2013, pp. 1085–1090.
- [15] J. S. Chase, A. J. Gallatin, and K. G. Yocum. "End System Optimizations for High-Speed TCP." In: *IEEE Communications Magazine* 39.4 (Apr. 2001), pp. 68–74.
- [16] Cisco Systems Inc. *Cisco VNI: Forecast and Methodology*. 2014. URL: [http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white\\_paper\\_c11-481360.pdf](http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white_paper_c11-481360.pdf).
- [17] T. C. Clancy. "Efficient OFDM Denial: Pilot Jamming and Pilot Nulling." In: *Proc. of the 2011 IEEE International Conference on Communications*. ICC '11. IEEE, 2011, pp. 1–5.
- [18] J. Classen, M. Schulz, and M. Hollick. "Practical Covert Channels for WiFi Systems." In: *Proc. of the 2015 IEEE Conference on Communications and Network Security*. CNS '15. IEEE, Sept. 2015, pp. 209–217.
- [19] J. Classen and D. Wegemer. *Leaking and Modifying Fitbit Data*. 2017. URL: <https://cfp.mrmcd.net/2017/talk/3T9E8Y/>.
- [20] F. C. Commission. *Jammer Enforcement*. 2017. URL: <https://www.fcc.gov/general/jammer-enforcement>.
- [21] *CortexTM-R4 and Cortex-R4F - Technical Reference Manual*. Revision: r1p4. ARM Limited. Apr. 2011.
- [22] E. Courjaud. *RF transmitter for Raspberry Pi*. URL: <https://github.com/F50E0/rpitx>.
- [23] B. DeBruhl, C. Kroer, A. Datta, T. Sandholm, and P. Tague. "Power Napping with Loud Neighbors: Optimal Energy-constrained Jamming and Anti-jamming." In: *Proc. of the 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks*. WiSec '14. ACM, 2014, pp. 117–128.
- [24] A. Dutta, D. Saha, D. Grunwald, and D. Sicker. "Secret Agent Radio: Covert Communication through Dirty Constellations." In: *Information Hiding*. Vol. 7692. Lecture Notes in Computer Science. Springer. 2013.
- [25] A. Engel and A. Koch. "Heterogeneous Wireless Sensor Nodes that Target the Internet of Things." In: *IEEE Micro* 36.6 (Nov. 2016), pp. 8–15.
- [26] B. Fang, N. D. Lane, M. Zhang, A. Boran, and F. Kawsar. "BodyScan: Enabling Radio-based Sensing on Wearable Devices for Contactless Activity and Vital Sign Monitoring." In: *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*. MobiSys '16. ACM, 2016, pp. 97–110.
- [27] L. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.
- [28] L. Frikha and Z. Trabelsi. "A new Covert Channel in WIFI Networks." In: *Proc. of the 3rd International Conference on Risks and Security of Internet and Systems*. CRiSYS '08. Oct. 2008, pp. 255–260.

- [29] C. G. Girling. "Covert Channels in LAN's." In: *IEEE Transactions on Software Engineering* SE-13.2 (1987), pp. 292–296.
- [30] S. Gollakota and D. Katabi. "Physical Layer Wireless Security Made Fast and Channel Independent." In: *Proc. of the 30th IEEE International Conference on Computer Communications*. INFOCOM '11. IEEE, Apr. 2011, pp. 1125–1133.
- [31] S. Gollakota, H. Hassanieh, B. Ransford, D. Katabi, and K. Fu. "They Can Hear Your Heartbeats: Non-invasive Security for Implantable Medical Devices." In: *Proc. of the 2011 ACM Conference of the Special Interest Group on Data Communication*. SIGCOMM '11. ACM, 2011, pp. 2–13.
- [32] S. Grabski and K. Szczypiorski. "Steganography in OFDM Symbols of Fast IEEE 802.11n Networks." In: *Proc. of the 2013 IEEE Security and Privacy Workshops*. SPW '13. IEEE, May 2013, pp. 158–164.
- [33] F. Gringoli and L. Nava. *OpenFWWF: Open FirmWare for WiFi networks*. 2009. URL: <http://netweb.ing.unibs.it/~openfwf/>.
- [34] D. Halperin, W. Hu, A. Sheth, and D. Wetherall. "Tool Release: Gathering 802.11n Traces with Channel State Information." In: *ACM SIGCOMM CCR* 41.1 (2011), p. 53.
- [35] B. Han, A. Schulman, F. Gringoli, N. Spring, B. Bhattacharjee, L. Nava, L. Ji, S. Lee, and R. R. Miller. "Maranello: Practical Partial Packet Recovery for 802.11." In: *Proc. of the 14th USENIX Symposium on Networked Systems Design and Implementation*. NSDI '10. USENIX Association, 2010, pp. 205–218.
- [36] M. Han, T. Yu, J. Kim, K. Kwak, and S. Lee. "OFDM Channel Estimation with Jammed Pilot Detector under Narrow-Band Jamming." In: *IEEE Transactions on Vehicular Technology* 57.3 (2008), pp. 1934–1939.
- [37] P. Helle, H. Lakshman, M. Siekmann, J. Stegemann, T. Hinz, H. Schwarz, D. Marpe, and T. Wiegand. "A Scalable Video Coding Extension of HEVC." In: *Proc. of the 2013 Data Compression Conference*. DCC '13. Mar. 2013, pp. 201–210.
- [38] "HICCUPS: Hidden Communication System for Corrupted Networks." In: *In Proc. of the 10th International Multi-Conference on Advanced Computer Systems*. ACS '03. 2003, pp. 31–40.
- [39] Z. Hijaz and V. S. Frost. "Exploiting OFDM Systems for Covert Communication." In: *Proc. of the 2010 Military Communications Conference*. MILCOM '10. IEEE, Oct. 2010, pp. 2149–2155.
- [40] R. Hirst. *PiFmDma Repository*. URL: <https://github.com/richardghirst/PiBits/tree/master/PiFmDma>.
- [41] J. Hoffmann. *Implementing a Mesh-Routing-Protokoll in the BCM4339 WiFi Chip*. Dipl. thesis. 2016.
- [42] T. D. Vo-Huu, T. D. Vo-Huu, and G. Noubir. "Interleaving Jamming in Wi-Fi Networks." In: *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks*. WiSec '16. ACM, 2016, pp. 31–42.
- [43] O. Ildis, Y. Ofir, and R. Feinstein. *bcmon blog*. 2012. URL: <http://bcmon.blogspot.de/>.

- [44] O. Ildis, Y. Ofir, and R. Feinstein. *Wardriving from your pocket — Using Wireshark to Reverse Engineer Broadcom WiFi chipsets*. 2013. URL: <https://recon.cx/2013/schedule/events/7.html>.
- [45] M. Jarschel, F. Wamser, T. Hohn, T. Zinner, and P. Tran-Gia. "SDN-Based Application-Aware Networking on the Example of YouTube Video Streaming." In: *Proc. of the 2nd European Workshop on Software Defined Networks*. EWSDN '13. IEEE, Oct. 2013, pp. 87–92.
- [46] A. Jeyaraj and M. E. Zarki. "A Real-Time Cross-Layer Design of the Multimedia Application Layer with a MIMO based Wireless Physical Layer." In: *Proc. of the 3rd International Symposium on Wireless Pervasive Computing*. ISWPC '08. IEEE, May 2008, pp. 455–458.
- [47] Z. Jiang, J. Zhao, X. Y. Li, J. Han, and W. Xi. "Rejecting the Attack: Source Authentication for Wi-Fi Management Frames using CSI Information." In: *Proc. of the 32nd IEEE International Conference on Computer Communications*. INFOCOM '13. IEEE, Apr. 2013, pp. 2544–2552.
- [48] M. L. Jorgensen, B. R. Yanakiev, G. E. Kirkelund, P. Popovski, H. Yomo, and T. Larsen. "Shout to Secure: Physical-Layer Wireless Security with Known Interference." In: *Proc. of the 2017 IEEE Global Telecommunications Conference*. GLOBECOM '07. IEEE, 2007, pp. 33–38.
- [49] P. Katz. *String searcher, and compressor using same*. US Patent 5,051,745. Sept. 1991.
- [50] Y. S. Kim, P. Tague, H. Lee, and H. Kim. "Carving Secure Wi-fi Zones with Defensive Jamming." In: *Proc. of the 7th ACM Symposium on Information, Computer and Communications Security*. ASIACCS '12. ACM, 2012, pp. 53–54.
- [51] M. Koch. "Reactive, Smartphone-based Jammer for IEEE 802.11 Networks." M.Sc. thesis. Technische Universität Darmstadt, Germany, 2016.
- [52] C. Krätzer, J. Dittmann, A. Lang, and T. Kühne. "WLAN Steganography: A First Practical Review." In: *Proc. of the 8th Workshop on Multimedia and Security*. MM&Sec '06. ACM, 2006, pp. 17–22.
- [53] S. Laga, T. V. Cleemput, F. V. Raemdonck, F. Vanhoutte, N. Bouten, M. Claeys, and F. D. Turck. "Optimizing Scalable Video Delivery through OpenFlow Layer-based Routing." In: *Proc. of the 2014 IEEE Network Operations and Management Symposium*. NOMS '14. IEEE, May 2014, pp. 1–4.
- [54] B. W. Lampson. "A Note on the Confinement Problem." In: *Commun. ACM* 16.10 (1973), pp. 613–615.
- [55] Y.-S. Li, C.-C. Chen, T.-A. Lin, C.-H. Hsu, Y. Wang, and X. Liu. "An End-to-End Testbed for Scalable Video Streaming to Mobile Devices over HTTP." In: *Proc. of the 2013 IEEE International Conference on Multimedia and Expo*. ICME '13. IEEE, July 2013, pp. 1–6.
- [56] X. Li, D. Zhang, Q. Lv, J. Xiong, S. Li, Y. Zhang, and H. Mei. "IndoTrack: Device-Free Indoor Human Tracking with Commodity Wi-Fi." In: *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 1.3 (Sept. 2017), 72:1–72:22.



- [57] Z. Li and T. He. "WEBee: Physical-Layer Cross-Technology Communication via Emulation." In: *Proc. of the 23rd ACM International Conference on Mobile Computing and Networking*. MobiCom '17. ACM, Oct. 2017, pp. 2–14.
- [58] G. Lin and G. Noubir. "On Link Layer Denial of Service in Data Wireless LANs: Research Articles." In: *Wireless Communications & Mobile Computing* 5.3 (May 2005), pp. 273–284.
- [59] J. Link. "Wi-Fi based Covert Channels on Android Smartphones." B.Sc. thesis. Technische Universität Darmstadt, Germany, 2017.
- [60] Mango Communications. *WARP Project*. 2017. URL: <http://warpproject.org>.
- [61] I. Martinovic, P. Pichota, and J. B. Schmitt. "Jamming for Good: A Fresh Approach to Authentic Communication in WSNs." In: *Proc. of the 2nd ACM Conference on Wireless Network Security*. WiSec '09. ACM, 2009, pp. 161–168.
- [62] D. Martins and H. Guyennet. "Attacks with Steganography in PHY and MAC Layers of 802.15.4 Protocol." In: *Proc. of the 5th International Conference on Systems and Networks Communications*. ICSNC '10. IEEE, Aug. 2010, pp. 31–36.
- [63] O. Mattos and O. Weigl. *Turning the Raspberry Pi Into an FM Transmitter*. URL: [http://www.icrobotics.co.uk/wiki/index.php/Turning\\_the\\_Raspberry\\_Pi\\_Into\\_an\\_FM\\_Transmitter](http://www.icrobotics.co.uk/wiki/index.php/Turning_the_Raspberry_Pi_Into_an_FM_Transmitter).
- [64] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. "OpenFlow: Enabling Innovation in Campus Networks." In: *ACM SIGCOMM Computer Communication Review* 38.2 (2008).
- [65] A. Mpitziopoulos, D. Gavalas, G. Pantziou, and C. Konstantopoulos. "Defending Wireless Sensor Networks from Jamming Attacks." In: *Proc. of the 18th International Symposium on Personal, Indoor and Mobile Radio Communications*. PIMRC '07. Sept. 2007, pp. 1–5.
- [66] S. Mühlbach and A. Koch. "A Reconfigurable Platform and Programming Tools for High-Level Network Applications Demonstrated as a Hardware Honey-pot." In: *IEEE Journal on Selected Areas in Communications* 32.10 (Oct. 2014), pp. 1919–1932.
- [67] Osmocom. *rtl-sdr*. URL: <https://osmocom.org/projects/sdr/wiki/rtl-sdr>.
- [68] K. Pelechrinis, M. Iliofotou, and S. V. Krishnamurthy. "Denial of Service Attacks in Wireless Networks: The Case of Jammers." In: *IEEE Communications Surveys & Tutorials* 13.2 (2011), pp. 245–257.
- [69] G. Peng, G. Zhou, D. T. Nguyen, and X. Qi. "All or None? The Dilemma of Handling WiFi Broadcast Traffic in Smartphone Suspend Mode." In: *Proc. of the 34th International Conference on Computer Communications*. INFOCOM '15. IEEE, Apr. 2015, pp. 1212–1220.
- [70] A. Proano and L. Lazos. "Selective Jamming Attacks in Wireless Networks." In: *Proc. of the 2010 IEEE International Conference on Communications*. ICC '10. IEEE, May 2010, pp. 1–6.
- [71] J. Rajahalme, A. Conta, B. Carpenter, and S. Deering. *IPv6 Flow Label Specification*. RFC 3697. 2004.

- [72] F. Ricciato, S. Sciancalepore, F. Gringoli, N. Facchi, and G. Boggia. "Position and Velocity Estimation of a Non-cooperative Source From Asynchronous Packet Arrival Time Measurements." In: *IEEE Transactions on Mobile Computing* PP.99 (2018), p. 1.
- [73] A. Saifullah, Y. Xu, C. Lu, and Y. Chen. "End-to-End Communication Delay Analysis in Industrial Wireless Networks." In: *IEEE Transactions on Computers* 64.5 (May 2015), pp. 1361–1374.
- [74] M. Schulz, F. Gringoli, D. Steinmetzer, M. Koch, and M. Hollick. "Massive Reactive Smartphone-based Jamming Using Arbitrary Waveforms and Adaptive Power Control." In: *Proc. of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. WiSec '17. ACM, 2017, pp. 111–121.
- [75] M. Schulz, J. Link, F. Gringoli, and M. Hollick. "Shadow Wi-Fi: Teaching Smartphones to Transmit Raw Signals and to Extract Channel State Information to Implement Practical Covert Channels over Wi-Fi." Accepted to appear in *Proc. of the 16th ACM International Conference on Mobile Systems, Applications, and Services*. 2018.
- [76] M. Schulz, D. Stohr, S. Wilk, B. Rudolph, W. Effelsberg, and M. Hollick. "APP and PHY in Harmony: A Framework Enabling Flexible Physical Layer Processing to Address Application Requirements." In: *Proc. of the 2015 International Conference and Workshops on Networked Systems*. NetSys '15. IEEE, Mar. 2015, pp. 1–8.
- [77] M. Schulz, D. Wegemer, and M. Hollick. "Nexmon: Build Your Own Wi-Fi Testbeds With Low-Level MAC and PHY-Access Using Firmware Patches on Off-the-Shelf Mobile Devices." In: *Proc. of the 11th Workshop on Wireless Network Testbeds, Experimental Evaluation & CHaracterization*. WiNTECH '17. ACM, Oct. 2017, pp. 59–66.
- [78] M. Schulz, E. Deligeorgopoulos, M. Hollick, and F. Gringoli. "DEMO: Demonstrating Reactive Smartphone-Based Jamming." In: *Proc. of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. WiSec '17. ACM, 2017, pp. 285–287.
- [79] M. Schulz, D. Wegemer, and M. Hollick. "NexMon: A Cookbook for Firmware Modifications on Smartphones to Enable Monitor Mode." In: *arXiv:1601.07077* (2015).
- [80] M. Schulz, D. Wegemer, and M. Hollick. *Nexmon: The C-based Firmware Patching Framework*. 2017. URL: <https://nexmon.org>.
- [81] H. Schwarz and M. Wien. "The Scalable Video Coding Extension of the H.264/AVC Standard." In: *IEEE Signal Processing Magazine* 25.2 (2008).
- [82] C. Shahriar, S. Sodagari, R. McGwier, and T. C. Clancy. "Performance Impact of Asynchronous Off-Tone Jamming Attacks against OFDM." In: *Proc. of the 2013 International Conference on Communications*. ICC '13. IEEE, June 2013, pp. 2177–2182.
- [83] W. Shen, P. Ning, X. He, and H. Dai. "Ally Friendly Jamming: How to Jam Your Enemy and Maintain Your Own Wireless Connectivity at the Same Time." In: *Proc. of the 2013 Symposium on Security and Privacy*. S&P '13. IEEE, May 2013, pp. 174–188.
- [84] *Single-Chip 5G WiFi IEEE 802.11ac MAC/Baseband/Radio with Integrated Bluetooth 4.1 and FM Receiver*. CYW4339. Document No. 002-14784 Rev. \*G. CYPRESS. Oct. 16.

- [85] D. Steinmetzer, D. Wegemer, M. Schulz, J. Widmer, and M. Hollick. "Compressive Millimeter-Wave Sector Selection in Off-the-Shelf IEEE 802.11ad Devices." In: *Proc. of the 13th International Conference on Emerging Networking EXperiments and Technologies*. CoNEXT '17. ACM, 2017, pp. 414–425.
- [86] G. Sullivan, J. Ohm, W.-J. Han, and T. Wiegand. "Overview of the High Efficiency Video Coding (HEVC) Standard." In: *IEEE Transactions on Circuits and Systems for Video Technology* 22.12 (2012).
- [87] I. Tinnirello, G. Bianchi, P. Gallo, D. Garlisi, F. Giuliano, and F. Gringoli. "Wireless MAC Processors: Programming MAC Protocols on Commodity Hardware." In: *Proc. of the 31st International Conference on Computer Communications*. INFOCOM '12. IEEE, Mar. 2012, pp. 1269–1277.
- [88] M. Vanhoef and F. Piessens. "Advanced Wi-Fi Attacks Using Commodity Hardware." In: *Proc. of the 30th Annual Computer Security Applications Conference*. ACSAC '14. ACM, 2014, pp. 256–265.
- [89] WARP Project – Throughput Benchmarks. 2014. URL: <http://warpproject.org/trac/wiki/802.11/Benchmarks/Throughput>.
- [90] D. Wegemer. "Energy Efficient WiFi Analysis Framework on Smartphones." M.Sc. thesis. Technische Universität Darmstadt, Germany, 2016.
- [91] M. Wilhelm, I. Martinovic, J. B. Schmitt, and V. Lenders. "Short Paper: Reactive Jamming in Wireless Networks: How Realistic is the Threat?" In: *Proc. of the 4th ACM Conference on Wireless Network Security*. WiSec '11. ACM, 2011, pp. 47–52.
- [92] M. Wilhelm, I. Martinovic, J. B. Schmitt, and V. Lenders. "WiFire: A Firewall for Wireless Networks." In: *Proc. of the ACM Conference of the Special Interest Group on Data Communication*. SIGCOMM '11. ACM, 2011, pp. 456–457.
- [93] M. Wolf. "Covert Channels in LAN Protocols." In: *Local Area Network Security: Workshop LANSEC '89*. Ed. by T. A. Berson and T. Beth. Springer Berlin Heidelberg, 1989, pp. 89–101.
- [94] Y. Xie, Z. Li, and M. Li. "Precise Power Delay Profiling with Commodity WiFi." In: *Proc. of the 21st Annual International Conference on Mobile Computing and Networking*. MobiCom '15. ACM, 2015, pp. 53–64.
- [95] F. Xu, Z. Qin, C. C. Tan, B. Wang, and Q. Li. "IMDGuard: Securing Implantable Medical Devices with the External Wearable Guardian." In: *Proc. of 30th IEEE Conference on Computer Communications*. INFOCOM '11. IEEE, Apr. 2011, pp. 1862–1870.
- [96] W. Xu, W. Trappe, Y. Zhang, and T. Wood. "The Feasibility of Launching and Detecting Jamming Attacks in Wireless Networks." In: *Proc. of the 6th ACM International Symposium on Mobile Ad Hoc Networking and Computing*. MobiHoc '05. ACM, 2005, pp. 46–57.
- [97] Q. Yan, H. Zeng, T. Jiang, M. Li, W. Lou, and Y. T. Hou. "MIMO-based Jamming Resilient Communication in Wireless Networks." In: *Proc. of the 33rd IEEE Conference on Computer Communications*. INFOCOM '14. IEEE, Apr. 2014, pp. 2697–2706.
- [98] Q. Yan, H. Zeng, T. Jiang, M. Li, W. Lou, and Y. T. Hou. "Jamming Resilient Communication using MIMO Interference Cancellation." In: *IEEE Transactions on Information Forensics and Security* 11.7 (July 2016).

- [99] K.-K. Yap, M. Kobayashi, D. Underhill, S. Seetharaman, P. Kazemian, and N. McKeown. "The Stanford OpenRoads Deployment." In: *Proc. of the 4th ACM International Workshop on Experimental Evaluation and Characterization*. WiNTECH '09. ACM, 2009, pp. 59–66.

## CURRICULUM VITÆ

---

### Personal Information

<i>Name</i>	Matthias Thomas Schulz
<i>Date of Birth</i>	November 15, 1987
<i>Place of Birth</i>	Wiesbaden-Dotzheim, Germany
<i>Nationality</i>	German

### Education

<i>since 05/2013</i>	<b>Doctoral candidate</b> Computer Science Technische Universität Darmstadt, Darmstadt, Germany
<i>09/2010 – 05/2013</i>	<b>Master of Science</b> Electrical Engineering and Information Technology (specialization: Computer Engineering) Technische Universität Darmstadt, Darmstadt, Germany Average Grade: 1.1
<i>10/2007 – 08/2010</i>	<b>Bachelor of Science</b> Electrical Engineering and Information Technology (specialization: Computer Engineering) Technische Universität Darmstadt, Darmstadt, Germany Average Grade: 1.4
<i>08/1998 – 03/2007</i>	<b>General Higher Education Entrance Qualification</b> Intensive Courses: Mathematics, Physics, and English Gymnasium am Kurfürstlichen Schloss in Mainz, Mainz, Germany Average Grade: 1.5

### Work Experience

<i>05/2013 – 02/2018</i>	<b>Research Associate</b> Secure Mobile Networking Lab (SEEMOO) Technische Universität Darmstadt, Darmstadt, Germany
<i>01/2009 – 05/2016</i>	<b>Software Developer and Server Administrator</b> mlgraphics.de (projected ended), Mainz-Kastel, Germany
<i>06/2011 – 10/2011</i>	<b>Intern during Master Studies</b> Rohde & Schwarz, Munich, Germany
<i>06/2007 – 08/2007</i>	<b>Intern before Bachelor Studies</b> Eckelmann AG, Wiesbaden, Germany

04/2007 – 05/2007 **Intern before Bachelor Studies**  
Hilscher Gesellschaft für Systemautomation, Hattersheim, Germany

## Awards

*Publication* **Best Paper Award at ACM WiNTECH 2017**  
Paper title: “Nexmon: Build Your Own Wi-Fi Testbeds With Low-Level MAC and PHY-Access Using Firmware Patches on Off-the-Shelf Mobile Devices”

*Publication* **Best Paper Award at ACM WiSec 2017**  
Paper title: “Massive Reactive Smartphone-Based Jamming using Arbitrary Waveforms and Adaptive Power Control”

*Teaching* **Athene Award for Good Teaching 2016**  
Special prize in the category study project for the research project in the “Physical Layer Security in Wireless Systems” course

*Publication* **Best Paper Award at ACM WiSec 2015**  
Paper title: “Lockpicking Physical Layer Key Exchange: Weak Adversary Models Invite the Thief”

*Publication* **Best Paper Award at IEEE NetSys 2015**  
Demo title: “APP and PHY in Harmony: A Framework Enabling Flexible Physical Layer Processing to Address Application Requirements”

*Thesis* **Datenlotsen Award 2014**  
For one of the top three Master theses in the fields of Computer Science, Mathematics and Engineering Management at the TU Darmstadt offered by the Datenlotsen Informationssysteme GmbH

*High School* **Award for Exemplary Attitude and Commitment in School 2007**  
Offered by Rhineland-Palatinate’s Ministry of Education, Economics, Youth, and Culture

## Supervised and on-going Diploma, Master, and Bachelor Theses

*M.Sc. Thesis* **Markus Brandt** “Acoustic Covert Channels for Mobile Phone Platforms”

*M.Sc. Thesis* **Abdullah Tahir** “Eavesdropping and Packet Injection in Ethernet”

*B.Sc. Thesis* **Benedikt Rudolph** “CLICK2WARP – Integrating the click modular router and the wireless open-access research platform for network-scale experiments”

*M.Sc. Thesis* **Daniel Steinmetzer** “Security Analysis of Physical Layer Key Exchange Mechanisms”

*M.Sc. Thesis* **Patrick Thomas Michael Klapper** “Infecting the Wire: Semi-Automatic Wireless Eavesdropping, Packet Injection and Reactive Jamming on Wired IEEE 802.3 Ethernet Networks”

<i>M.Sc. Thesis</i>	<b>Daniel Wegemer</b> "Energy Efficient WiFi Analysis Framework on Smartphones"
<i>Dipl. Thesis</i>	<b>Justus Hoffmann</b> "Implementing a Mesh-Routing-Protokoll in the BCM4339 WiFi Chip"
<i>B.Sc. Thesis</i>	<b>Victor Pecanins</b> "Development of a Low-Cost Software-Defined Radio with 2.4 GHz Tranceiver"
<i>M.Sc. Thesis</i>	<b>Michael Palm</b> "Secure Localization and Distance Bounding with IEEE 802.11"
<i>B.Sc. Thesis</i>	<b>Florentin Putz</b> "Probe request tracking in WiFi firmware"
<i>M.Sc. Thesis</i>	<b>Michael Koch</b> "Reactive IEEE 802.11 Jammers on Mobile Smartphone Platforms"
<i>M.Sc. Thesis</i>	<b>Fabian Knapp</b> "Nexmon-based Wireless Penetration Testing Suite for Android"
<i>M.Sc. Thesis</i>	<b>Carsten Bruns</b> "Modification of LTE Firmwares on Smartphones"
<i>M.Sc. Thesis</i>	<b>Robin Morawetz</b> "Self-Replicating Malwares for Wi-Fi Chips"
<i>B.Sc. Thesis</i>	<b>Jakob Link</b> "Wi-Fi based Covert Channels on Android Smartphones"
<i>B.Sc. Thesis</i>	<b>Nicolas Schickert</b> "Decompilation and Analysis of b43 Assembly Code used in Broadcom WiFi Chips"
<i>B.Sc. Thesis</i>	<b>Filip Lüneberg</b> "Implementing a WiFi Jammer on a Raspberry Pi"
<i>B.Sc. Thesis</i>	<b>Damir Mehmedovic</b> "Wi-Fi based Key Exchange on Android Smartphones"
<i>B.Sc. Thesis</i>	<b>Dennis Mantz</b> "Hacking Bluetooth Firmware of WiFi Combo Chips in Smartphones"

## Miscellaneous

<i>Organization</i>	Organization of the Security & Privacy Week 2016, Darmstadt, Germany
<i>Organization</i>	Finance Chair and Local Organization Chair of the 9th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec 2016)
<i>Organization</i>	Co-organization of the SEEMOO Signal Intelligence Challenge, 2016
<i>Organization</i>	Co-organization of the 2nd IEEE Signal Intelligence Challenge, 2015
<i>Teaching</i>	HDA Zertifikat Hochschullehre, 2015
<i>Teaching</i>	Creation and teaching of "Physical Layer Security in Wireless Systems (PhySec)" since winter semester 2013/14 (with teaching permit)
<i>Teaching</i>	Co-teaching of "Drahtlose Netze zur Krisenbewältigung (CriCom)" since winter semester 2014/15

*Teaching*                      Offering and supervising topics and experiments of “Secure Mobile Networking Lab and Project”, “Seminar on Networking, Security, Mobility, and Wireless Communications”, “Seminar on Security in Ad hoc, Sensor, and Mesh Networks (SWMN)”, and “Praktikum Kommunikationstechnik und Sensorsysteme (PKS)”

Kriftel, March 3, 2018



## AUTHOR'S PUBLICATIONS

---

### PRIMARY AND CO-FIRST AUTHOR

*Asterisks (\*) indicate co-first authorship.*

1. M. Schulz, D. Wegemer, and M. Hollick. **The Nexmon Firmware Analysis and Modification Framework: Empowering Researchers to Enhance Wi-Fi Devices**, accepted to appear in *Elsevier's Computer Communications Journal*, 2018.
2. M. Schulz, J. Link, F. Gringoli, and M. Hollick. **Shadow Wi-Fi: Teaching Smartphones to Transmit Raw Signals and to Extract Channel State Information to Implement Practical Covert Channels over Wi-Fi**, accepted to appear in *Proceedings of the 16th ACM International Conference on Mobile Systems, Applications, and Services (MobiSys 2018)*, June 2018.
3. M. Schulz, D. Wegemer, M. Hollick. **Nexmon: Build Your Own Wi-Fi Testbeds With Low-Level MAC and PHY-Access Using Firmware Patches on Off-the-Shelf Mobile Devices**, in *Proceedings of the 11th ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation & Characterization (WiNTECH 2017)*, October 2017.
4. M. Schulz, F. Knapp, E. Deligeorgopoulos, D. Wegemer, F. Gringoli, M. Hollick. **DEMO: Nexmon in Action: Advanced Applications Powered by the Nexmon Firmware Patching Framework**, in *Proceedings of the 11th ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation & Characterization (WiNTECH 2017)*, October 2017.
5. M. Schulz, F. Gringoli, D. Steinmetzer, M. Koch, M. Hollick. **Massive Reactive Smartphone-Based Jamming using Arbitrary Waveforms and Adaptive Power Control**, in *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec 2017)*, July 2017.
6. M. Schulz, E. Deligeorgopoulos, M. Hollick, F. Gringoli. **DEMO: Demonstrating Reactive Smartphone-Based Jamming**, in *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec 2017)*, July 2017.
7. M. Schulz, P. Klapper, M. Hollick, E. Tews and S. Katzenbeisser. **Trust The Wire, They Always Told Me! On Practical Non-Destructive Wire-Tap Attacks Against Ethernet**, in *Proceedings of the 9th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec 2016)*, July 2016.
8. M. Schulz, A. Loch, M. Hollick. **DEMO: Demonstrating Practical Known-Plaintext Attacks against Physical Layer Security in Wireless MIMO Systems**, in *Proceedings of the 9th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec 2016)*, July 2016.

9. M. Schulz, D. Wegemer, M. Hollick. **DEMO: Using NexMon, the C-based WiFi firmware modification framework**, in *Proceedings of the 9th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec 2016)*, July 2016.
10. J. Classen\*, M. Schulz\*, and M. Hollick. **Practical Covert Channels for WiFi Systems**, in *Proceedings of the IEEE Conference on Communications and Network Security (CNS 2015)*, September 2015.
11. D. Stohr\*, M. Schulz\*, M. Hollick and W. Effelsberg. **APP and PHY in Harmony: Demonstrating Scalable Video Streaming Supported by Flexible Physical Layer Control**, in *Proceedings of the International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM 2015)*, June 2015.
12. M. Schulz, D. Stohr, S. Wilk, B. Rudolph, W. Effelsberg and M. Hollick. **APP and PHY in Harmony: A Framework Enabling Flexible Physical Layer Processing to Address Application Requirements**, in *Proceedings of the International Conference on Networked Systems (NetSys 2015)*, March 2015.
13. Y. Zheng\*, M. Schulz\*, W. Lou, Y. T. Hou and M. Hollick. **Highly Efficient Known-Plaintext Attacks against Orthogonal Blinding based Physical Layer Security**, *IEEE Wireless Communications Letters*, vol. 4, no. 1, pp. 34-37, February 2015.
14. Matthias Schulz, Adrian Loch, and Matthias Hollick. **Practical Known-Plaintext Attacks against Physical Layer Security in Wireless MIMO Systems**, in *Proceedings of the Network and Distributed System Security Symposium (NDSS 2014)*, February 2014.

## CO-AUTHOR

15. D. Steinmetzer, D. Wegemer, M. Schulz, J. Widmer, M. Hollick. **Compressive Millimeter-Wave Sector Selection in Off-the-Shelf IEEE 802.11ad Devices**, in *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies (CoNEXT 2017)*, December 2017.
16. Y. Zheng, M. Schulz, W. Lou, Y. T. Hou and M. Hollick. **Profiling the Strength of Physical-Layer Security: A Study in Orthogonal Blinding**, in *Proceedings of the 9th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec 2016)*, July 2016.
17. I. E. Bagci, U. Roedig, I. Martinovic, M. Schulz and M. Hollick. **Using Channel State Information for Tamper Detection in the Internet of Things**, in *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC 2015)*, December 2015.
18. D. Steinmetzer, M. Schulz and M. Hollick. **Lockpicking Physical Layer Key Exchange: Weak Adversary Models Invite the Thief**, in *Proceedings of the 8th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec 2015)*, June 2015.
19. M. Maass, U. Müller, T. Schons, D. Wegemer and M. Schulz. **NFCGate: An NFC Relay Application for Android**, in *Proceedings of the 8th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec 2015)*, June 2015.

20. A. Loch, M. Schulz, M. Hollick. **Demo: WARP Drive - Accelerating Wireless Multi-hop Cross-layer Experimentation on SDRs**, in *Proceedings of the ACM SIGCOMM Software Radio Implementation Forum (SRIF 2014)*, August 2014.
21. I. E. Bagci, U. Roedig, M. Schulz and M. Hollick. **Short Paper: Gathering Tamper-Evidence in Wi-Fi Networks Based on Channel State Information**, in *Proceedings of the 7th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec 2014)*, July 2014.
22. M. Wichtlhuber, J. Rückert, D. Stingl, M. Schulz and D. Hausheer. **Energy-Efficient Mobile P2P Video Streaming**, in *Proceedings of the 12th IEEE International Conference on Peer-to-Peer Computing (P2P 2012)*, September 2012.

#### NON-ACADEMIC PUBLICATIONS

23. M. Schulz. **Nexmon – Wie man die eigene WLAN-Firmware hackt**, c't 26/2016, pp. 168, Heise Verlag, December 2016.
24. M. Schulz. **Praktikumsübung zur Implementierung des Radio Data Systems (RDS) auf Software Defined Radios**, *Virtuelle Instrumente in der Praxis 2014 – Begleitband zum 19. VIP-Kongress*, VDE Verlag, 2014.



## PREVIOUSLY PUBLISHED OR SUBMITTED MATERIAL

---

This thesis includes material previously published as peer-reviewed publications or currently accepted for publication. In accordance with the regulations of the Computer Science department at the Technische Universität Darmstadt, below, we list the sections by chapter which include verbatim fragments from these publications. The list was updated on May 26, 2018 after receiving the last acceptance notification.

### CHAPTER 1 – MOTIVATION AND GOALS

Chapter 1 revises Section 1 from [6].

### CHAPTER 3 – RELATED WORK

The introduction of Chapter 3 and Sections 3.1 and 3.2 revise Section 2 from [6].

### CHAPTER 4 – BROADCOM’S WI-FI CHIPS

Section 4.1 revises Section 3.1 from [6]. Section 4.2 revises Section 2 from [5]. Section 4.2.1 revises Section 3.2.1 from [6]. Section 4.2.2 revises Section 3.2.2 from [6]. Section 4.2.3 revises Section 3.2 from [3]. Section 4.3 revises Section 3.3 from [6]. Section 4.3.1 revises Section 3.2 from [3]. Section 4.3.2 revises Section 3.1 from [3]. Section 4.3.3 revises Section 3.3.3 from [6]. Section 4.4 revises Section 3.4 from [6]. Section 4.4.1 revises Section 3.4.1 from [6]. Section 4.5 revises Section 3.5 from [6]. Section 4.5.1 revises Section 3.5.1 from [6]. Section 4.5.2 revises Section 3.5.2 from [6].

### CHAPTER 5 – FIRMWARE ANALYSIS AND SECURITY IMPROVEMENTS

The introduction of Chapter 5 revises Section 4 from [6]. Section 5.1 revises Section 4.1 from [6]. Section 5.1.1 revises Section 4.1.1 from [6]. Section 5.2 revises Section 4.2 from [6]. Section 5.2.1 revises Section 4.2.1 from [6]. Section 5.2.2 revises Section 4.2.2 from [6]. Section 5.2.3 revises Section 4.2.3 from [6]. Section 5.2.4 revises Section 4.2.4 from [6]. Section 5.2.5 revises Section 4.2.5 from [6]. Section 5.2.6 revises Section 4.2.6 from [6]. Section 5.2.7 revises Section 4.2.7 from [6]. Section 5.2.8 revises Section 4.2.8 from [6]. Section 5.2.9 revises Section 4.2.9 from [6]. Section 5.2.10 revises Section 4.2.10 from [6].

### CHAPTER 6 – NEXMON FIRMWARE PATCHING FRAMEWORK

The introduction of Chapter 6 revises Section 1 from [5]. Section 6.1 revises Section 3 from [5]. Section 6.1.1 revises Section 3.1 from [5]. Section 6.1.2 revises Section 3.2 from [5]. Section 6.1.3 revises Section 3.3 from [5]. Section 6.1.4 revises Section 5.1.4 from [6]. Section 6.1.5 revises Section 3.4 from [5]. Section 6.1.6 revises Section 5.1.6 from [6]. Sec-

tion 6.1.7 revises Section 3.5 from [5]. Section 6.2 revises Section 4 from [5]. Section 6.2.1 revises Section 4.1 from [5]. Section 6.2.2 revises Section 4.2 from [5]. Section 6.2.3 revises Section 5.2.3 from [6]. Section 6.2.4 revises Section 4.3 from [5]. Section 6.2.5 revises Section 4.4 from [5]. Section 6.2.6 revises Section 4.5 from [5]. Section 6.2.7 revises Section 4.6 from [5] and Section 5.2.7 from [6]. Section 6.2.8 revises Section 5.2.8 from [6]. Section 6.2.9 revises Section 5.2.9 from [6]. Section 6.2.10 revises Section 5.2.10 from [6]. Section 6.2.11 revises Section 5.2.11 from [6]. Section 6.2.12 revises Section 5.2.12 from [6]. Section 6.2.13 revises Section 4.7 from [5] and Section 5.2.13 from [6]. Section 6.2.14 revises Section 4.8 from [5]. Section 6.2.15 revises Section 4.9 from [5]. Section 6.3 revises Section 6 from [5]. Section 6.4 revises Section 7 from [5].

## CHAPTER 7 – PROGRAMMABLE FIRMWARE DEBUGGER

The introduction of Chapter 7 revises Section 6 from [6]. Section 7.1 revises Section 6.1 from [6]. Section 7.2 revises Section 6.2 from [6]. Section 7.2.1 revises Section 6.2.1 from [6]. Section 7.2.2 revises Section 6.2.2 from [6]. Section 7.2.3 revises Section 6.2.3 from [6]. Section 7.2.4 revises Section 6.2.4 from [6]. Sections 7.3, and 7.4 revise Section 6.3 from [6].

## CHAPTER 8 – CHANNEL STATE INFORMATION EXTRACTOR

Sections 8.1, 8.1.1, and 8.1.2 revise Section 5.2 from [3]. Section 8.2 revises Section 6 from [3]. Sections 8.2.1, and 8.2.2 revise Section 6.3 from [3]. Section 8.3 revises Section 7 from [3].

## CHAPTER 9 – SOFTWARE-DEFINED RADIOS ON WI-FI CHIPS

The introduction of Chapter 9 revises Section 1, and Section 3.2 from [3]. Sections 9.1, and 9.1.1 revise Sections 3.3, and 5.1 from [3]. Section 9.2 revises Section 6 from [3]. Sections 9.2.1, and 9.2.2 revise Section 6.1 from [3]. Section 9.3 revises Section 7 from [3].

## CHAPTER 10 – PING OFFLOADING

The introduction of Chapter 10 revises Section 5 from [5]. Section 10.1 revises Section 5 from [5]. Section 10.2 revises Section 5 from [5]. Sections 10.2.1, 10.2.2, and 10.2.3 revise Section 5 from [5]. Section 10.3 revises Section 6 from [5].

## CHAPTER 11 – SDWNS WITH FLOW-BASED PHY CONTROL

The introduction of Chapter 11 revises Section 1 from [4]. Sections 11.1.1, 11.1.2, 11.1.3, 11.1.4, 11.1.5, 11.1.6, 11.1.7 revise Section 2 from [4]. Section 11.1.8 revises Section 3 from [4]. Section 11.2 revises Section 4 from [4]. Section 11.2.2 revises Section 3 from [7]. Section 11.3 revises Section 4.A from [4]. Section 11.3.1 revises Section 4.B from [4]. Section 11.3.2 revises Section 4.C from [4]. Section 11.3.3 revises Section 4.D from [4].

Section 11.4 revises Section 5 from [4]. Section 11.6 revises Section 6 from [4]. Section 11.7 revises Section 7 from [4].

## CHAPTER 12 – REACTIVE WI-FI JAMMING ON SMARTPHONES

The introduction of Chapter 12 revises Section 1 and Section 3.1 from [2]. Section 12.1 revises Section 4 from [2]. Section 12.1.1 revises Section 4.1 from [2]. Section 12.1.2 revises Section 4.2 from [2]. Section 12.1.3 revises Section 4.3 from [2]. Section 12.1.4 revises Section 3.2 from [2]. Section 12.1.5 revises Section 3.3 from [2]. Section 12.1.6 revises Section 3.4 from [2]. Section 12.2 revises Footnote 3 from [2]. Section 12.3 revises Section 5 from [2]. Section 12.3.1 revises Section 5.1 from [2]. Section 12.3.2 revises Section 5.2 from [2]. Section 12.3.3 revises Section 5.3 from [2]. Section 12.3.4 revises Section 5.4 from [2]. Section 12.3.5 revises Section 5.5 from [2]. Section 12.3.6 revises Section 5.6 from [2]. Section 12.4 revises Section 6 from [2]. Section 12.5 revises Section 2 from [2]. Section 12.7 revises Section 7 from [2].

## CHAPTER 13 – WI-FI-BASED COVERT CHANNELS

The introduction of Chapter 13 revises Section 1 from [3]. Section 13.1 revises Section 4 and Section 5.3 from [3]. Section 13.2, and 13.2.1 revise Section 5.3 from [3]. Section 13.2.2 revises Section 6.4 from [3]. Section 13.3 revises Section 6 and 6.4 from [3]. Sections 13.3.1, 13.3.2, 13.3.3, 13.3.4, 13.3.5, and 13.3.6 revise Section 6.4 from [3]. Section 13.4 revises Section 7 from [3]. Section 13.5 revises Section 2 from [3] and Section 6 from [1]. Sections 13.5.1, and 13.5.2 revise Section 6 from [1].

## TEXT SOURCES

- [1] J. Classen, M. Schulz, and M. Hollick. “Practical Covert Channels for WiFi Systems.” In: *Proc. of the 2015 IEEE Conference on Communications and Network Security*. CNS ’15. IEEE, Sept. 2015, pp. 209–217.
- [2] M. Schulz, F. Gringoli, D. Steinmetzer, M. Koch, and M. Hollick. “Massive Reactive Smartphone-based Jamming Using Arbitrary Waveforms and Adaptive Power Control.” In: *Proc. of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. WiSec ’17. ACM, 2017, pp. 111–121.
- [3] M. Schulz, J. Link, F. Gringoli, and M. Hollick. “Shadow Wi-Fi: Teaching Smartphones to Transmit Raw Signals and to Extract Channel State Information to Implement Practical Covert Channels over Wi-Fi.” Accepted to appear in *Proc. of the 16th ACM International Conference on Mobile Systems, Applications, and Services*. 2018.
- [4] M. Schulz, D. Stohr, S. Wilk, B. Rudolph, W. Effelsberg, and M. Hollick. “APP and PHY in Harmony: A Framework Enabling Flexible Physical Layer Processing to Address Application Requirements.” In: *Proc. of the 2015 International Conference and Workshops on Networked Systems*. NetSys ’15. IEEE, Mar. 2015, pp. 1–8.

- [5] M. Schulz, D. Wegemer, and M. Hollick. "Nexmon: Build Your Own Wi-Fi Testbeds With Low-Level MAC and PHY-Access Using Firmware Patches on Off-the-Shelf Mobile Devices." In: *Proc. of the 11th Workshop on Wireless Network Testbeds, Experimental Evaluation & CHaracterization*. WiNTECH '17. ACM, Oct. 2017, pp. 59–66.
- [6] M. Schulz, D. Wegemer, and M. Hollick. "The Nexmon Firmware Analysis and Modification Framework: Empowering Researchers to Enhance Wi-Fi Devices." Accepted to appear in Elsevier Computer Communications (COMCOM) Journal. 2018.
- [7] D. Stohr, M. Schulz, M. Hollick, and W. Effelsberg. "APP and PHY in Harmony: Demonstrating Scalable Video Streaming Supported by Flexible Physical Layer Control." In: *Proc. of the 2015 International Symposium on A World of Wireless, Mobile and Multimedia Networks*. WoWMoM '15. IEEE, June 2015, pp. 1–3.



## ERKLÄRUNG ZUR DISSERTATIONSSCHRIFT

---

*gemäß § 9 der Allgemeinen Bestimmungen der Promotionsordnung der  
Technischen Universität Darmstadt vom 12. Januar 1990 (ABl. 1990, S. 658)  
in der Fassung der VII. Änderung vom 28. September 2010*

Hiermit versichere ich die vorliegende Dissertationsschrift ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Eigenzitate aus vorausgehenden wissenschaftlichen Veröffentlichungen werden in Anlehnung an die Hinweise des Promotionsausschusses FB Informatik zum Thema „Eigenzitate in wissenschaftlichen Arbeiten“ (EZ-2014/10) in Kapitel „Previously Published or Submitted Material“ auf Seite 195 ff. gelistet. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen. In der abgegebenen Dissertationsschrift stimmen die schriftliche und die elektronische Fassung überein.

*Darmstadt, 12. Januar 2018*

---

Matthias Thomas Schulz